Erstprüfer/in: Prof. Dr. Ansgar Gerlicher  
Zweitprüfer/in: Sebastian Gerber

**Bachelorarbeit**

im Studiengang  
Medieninformatik

vorgelegt von

**Malte Leon Lohrer**Matr.-Nr.: 27399

am 05. April 2018   
an der Hochschule der Medien Stuttgart

Continuous Integration der Diagnosetoolkette der Daimler AG

# Ehrenwörtliche Erklärung

„Hiermit versichere ich, Malte Leon Lohrer ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: „Continuous Integration der Diagnosetoolkette der Daimler AG“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

01.04.2018

# Kurzfassung

# Abstract

# Inhaltsverzeichnis

Ehrenwörtliche Erklärung 2

Kurzfassung 3

Abstract 3

Inhaltsverzeichnis 4

Abbildungsverzeichnis 6

Abkürzungsverzeichnis 7

Vorwort 8

1 Überblick 9

2 Ziele 10

3 Continuous Integration 12

3.1 Das Konzept der Continuous Integration und deren Vorteile 12

3.1.1 Gemeinsame Codebasis 13

3.1.2 Automatisierter Build 14

3.1.3 Selbsttestender Build 14

3.1.4 Häufige Integration 15

3.1.5 Builds und Tests nach jeder Änderung 15

3.1.6 Schnelle Build-Zyklen 17

3.1.7 Tests in gespiegelter Produktionsumgebung 18

3.1.8 Einfacher Zugriff auf Build-Ergebnisse 20

3.1.9 Automatisierte Berichte 20

3.1.10 Automatisierte Verteilung 20

3.2 Nachteile der CI 21

4 Eingeschlagener Realisierungsweg 23

4.1 Verwendete Tools, Softwaretechnologien und Datenstrukturen 23

4.1.1 Python als Programmiersprache 23

4.1.2 Die Python Distribution *Anaconda* 24

4.1.3 PyCharm als Entwicklungsumgebung 25

4.1.4 *Git* im Zusammenspiel mit *TortoiseGit* und *GitLab* als Versionskontrollsystem 25

4.1.5 Testing mit Unit Tests 29

4.1.6 GUI automatisierung mit pywinauto 30

4.1.7 Luigi 30

4.1.8 DTS Monaco 33

4.1.9 Wichtige Datenstrukturen 33

4.2 Das automatisierte Testen des Diagnosetools DTS Monaco 35

4.2.1 Das Erstellen der Basis-Verzeichnisse 36

4.2.2 Das Einloggen in das Diagnoseportal 37

4.2.3 Das Downloaden und parsen der Metaview 38

4.2.4 Das Erstellen der ECU Verzeichnisse innerhalb des Working Directorys 39

4.2.5 Download der Diagnosedateien aus dem Diagnoseportal 40

4.2.6 DTS Monaco automatisiert Starten 42

4.2.7 Die PDX-Datei Entpacken 43

4.2.8 Informationen aus der ODX-Datei herausziehen 44

4.2.9 Die Simulationsdatei mit den richtigen Daten Befüllen 44

4.2.10 Die SMR-D-Datei in das dbr-Verzeichnis von DTS Monaco kopieren 45

4.2.11 Start der Simulation 45

4.2.12 Schließen des DTS Monaco Workspaces 46

4.2.13 Der Vergleich der Kurztestergebnisse mit den Vorgaben der Simulation 46

4.2.14 Das Bündeln aller Tasks und das anschließende Starten des gesamten Testvorgangs 48

Quellenverzeichnis 50

# Abbildungsverzeichnis

# Abkürzungsverzeichnis

CI *Continuous Integration*

GUI *Graphical User Interface*

VCS *Version control system*

DVCS *Distributed* *Version control system*

API *Application Programming Interface*

URL *Uniform Resource Locator*

ECU *Electronic Control Unit*

DP Diagnoseportal

# Vorwort

# Überblick

# Ziele

Ziel der vorliegenden Arbeit ist es, das Konzept der *Continuous Integration* genau zu beleuchten und theoretischauf die Diagnosetoolkette der Daimler AG anzuwenden. Die dabei gewonnenen Erkenntnisse werden die Vor- und Nachteile einer zukünftigen Entwicklung der Diagnose-Toolkette mithilfe von CI aufzeigen.

Der praktische Teil der Thesis, welcher auf der beiliegenden CD enthalten ist, stellt einen ersten Schritt in Richtung CI der Diagnose-Toolkette dar. Ziel dieses Teils ist es, das Diagnosetool *DTS* *Monaco* automatisiert zu testen, um damit dessen Funktionalität zuverlässig gewährleisten zu können. *DTS* *Monaco* ist eine Komponente innerhalb der Diagnose-Toolkette und wird über eine komplexe *GUI* gesteuert, welche im Zuge dessen ebenfalls automatisiert getestet werden muss.

Nachfolgend sind die Feinziele der Praktischen Arbeit aufgelistet:

1. Gemeinsame Codebasis (erfüllt)
   * Gerade im Hinblick auf eine CI ist es enorm wichtig direkt von Anfang an die Möglichkeit einer gemeinsamen Codebasis zu schaffen. Diese wird durch ein *GitLab Repository* realisiert, welches in Zukunft problemlos von anderen Teams genutzt werden kann.
2. Modularer Code (erfüllt)
   * Ein weiteres Ziel ist es den Quellcode in möglichst kleine Module aufzuteilen. Das bringt nicht nur durch eine bessere Übersicht Vorteile, sondern kommt auch der Wartbarkeit des Quellcodes enorm entgegen. Außerdem lassen sich kleinere Teile Quellcode deutlich leichter, effizienter und vor allem abgekapselt testen, was bei einer guten CI eine große Rolle spielt.
3. Automatisierung der grafischen Benutzeroberfläche von *DTS Monaco (erfüllt)*
   * Ein wesentliches Ziel besteht darin, das Tool *DTS Monaco*, auf welche in Kapitel <X> genauer eingegangen wird, zu automatisieren. Da diese Anwendung über eine Grafische Benutzeroberfläche (im Folgenden als „GUI“ bezeichnet) gesteuert wird, müssen hierbei die Aktionen die normalerweise ein Mensch durchführt, vom Code übernommen werden.
4. Bereitstellen von Unit Tests (erfüllt)
   * Unverzichtbar für eine richtig durchgeführte CI ist es, genügend Tests zu implementieren, welche stets die Funktion der einzelnen Codebestandteile sicherstellen. Hierbei werden sogenannte *Unit Tests* durchgeführt, welche jeder für sich nur sehr kleine des Quellcodes testen, dafür jedoch sehr schnell und präzise.
5. Automatisches generieren der Ergebnisse (erfüllt)
   * Die Nachvollziehbarkeit des Programmdurchlaufs stellt ein weiteres Ziel dar. Diese wird durch das protokollieren der Ergebnisse der einzelnen Teilabschnitte eines jeden Durchlaufs gewährleistet. Nach einem vollständigen Programmablauf werden Daten bereitgestellt, welche genau aufzeigen welche Funktionen zu welchem Zeitpunkt abgelaufen sind und ob es dabei zu Fehlern kam.
6. Abhängigkeiten zwischen den Funktionen klar definieren (erfüllt)
   * Es muss gewährleistet sein, dass bestimmte Teile des Codes erst dann ausgeführt werden, wenn andere Teile schon erfolgreich durchlaufen wurden. Diese Abhängigkeiten müssen klar definiert werden. Wenn eine Funktion fehlschlägt, werden nachfolgende erst gar nicht ausgeführt. Diese Abhängigkeiten wurden mithilfe von *Luigi* realisiert, ein Python Modul auf welches in Kapitel 4.1.6 genauer eingegangen wird.
7. Visualisierung des gesamten Programmablaufs (erfüllt)
   * Eines der bedeutendsten Ziele stellt die Visualisierung des Programmablaufs dar. Wie schon bei der Definition der Abhängigkeiten, wird hier das Python Modul *Luigi* genutzt, um stets den Status des Ablaufs in grafisch aufbereiteter Form auf einer Website nachvollziehen zu können.

# Continuous Integration

Dieses Kapitel befasst sich mit der *Continuous Integration* als Solche und gibt dem Leser einen Einblick in die Softwareentwicklung unter Anwendung von CI.

Hierfür werden die Grundprinzipien einer CI erläutert und es wird auf deren Vor- und Nachteile eingegangen.

## Das Konzept der Continuous Integration und deren Vorteile

Die *Continuous Integration* ist nach Wiest (2010, S.13) eine Softwareentwicklungspraktik, welche erstmals als eine der Praktiken der sogenannten Extremprogrammierung (*eXtreme Programming*) populär wurde.

Der Begriff *Continuous Integration* entspringt dem gleichnamigen Artikel von Martin Fowler, welcher das erste Mal im Jahr 2000 erschien und im Jahr 2006 noch einmal überarbeitet und aktualisiert wurde.

In eben diesem Artikel beschreibt Fowler (2006) das Grundkonzept der CI folgendermaßen:

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.”

Grundsätzlich geht es bei CI nach Fowler darum, dass die Softwareentwickler die Änderungen welche sie am Code vornehmen so oft wie möglich integrieren, um die Deltas zwischen der bisherigen und der neuen Version möglichst gering zu halten. Der neue Softwarestand wird dann umgehend gebaut, wodurch Fehler, durch die kleinere Menge an Änderungen im Vergleich zum letzten Softwarestand, viel schneller gefunden werden können. Dadurch soll der Integrationsvorgang selbst zu einem *Non-Event* (dt.: nicht-Ereignis) werden. (Martin Fowler, 2006)

Fowler formulierte in seinem Artikel „*Continuous Integration*“ außerdem insgesamt 10 Praktiken, welche für eine effektive CI maßgeblich sind.

Für die Namen dieser Praktiken werden im Folgenden die deutschen Übersetzungen von Dr. Simon Wiest (2010: 15f) verwendet.

### Gemeinsame Codebasis

Sämtliche Daten, welche zu einem Softwareprojekt gehören, müssen in einem für alle an dem Projekt beteiligten Akteure offen und jederzeit zugänglich sein. Meist ist hier die Rede von einem Versionskontrollsystem (VCS, engl.: *version control system*). Versionskontrollsysteme gibt es heutzutage zu genüge. Einige der bekanntesten Vertreter sind u.a. *Git*, *Subversion(SVN)* oder auch *Mercurial*.

Es sei an dieser Stelle angemerkt, dass zwischen drei verschiedenen Arten der Versionskontrollverwaltung unterschieden wird: *Lokale* Versionskontrollsysteme, *Zentralisierte* Versionskontrollsysteme und *Verteilte* Versionskontrollsysteme (Git 19. Oktober 2017, Seite „Getting started - About version control“). Hier und im Folgenden, sind stets verteilte Versionskontrollsysteme (DVCS, engl.: *distributed version control system*) gemeint, wenn von einem Versionskontrollsystem (VCS) die Rede ist.

Die Hauptaufgabe eines VCS besteht darin, jede Änderung an einer Datei genau zu dokumentieren und nachvollziehbar zu machen. Diese Änderungen sind im Nachhinein auf einen genauen Zeitpunkt und auch auf die Person, welche die Änderung vorgenommen hat, zurückzuführen. Dadurch wird paralleles Arbeiten innerhalb desselben Projekts, bzw. sogar innerhalb identischer Dateien, ermöglicht. Entwickler arbeiten, wenn ein VCS benutzt wird, normalerweise auf ihrem eigenen *Branch*[[1]](#footnote-1)und führen einen *commit*[[2]](#footnote-2)aus, sobald sie eine Teilaufgabe abgeschlossen haben und bereit sind, diesen neuen Softwarestand in das VCS zu übertragen. Manchmal wird hier auch von einem *Check-In* gesprochen, was im Endeffekt aber im Bezug auf ein VCS dieselbe Bedeutung hat. Auf diese Weise können mehrere Entwickler parallel auf ihren jeweiligen *Branches* arbeiten. Zu einem beliebigen späteren Zeitpunkt lassen sich diese Branches zusammenführen. Dies ist eine weitere Aufgabe, die ein VCS übernimmt. Es ist dann die Rede von einem *Merge*, also zu Deutsch einer „Verschmelzung“ von zwei oder mehreren Branches. Es kann dabei zu Konflikten kommen, wenn während des parallelen Entwickelns identische Zeilen des Sourcecodes von mehreren Entwicklern modifiziert wurden. Das VCS benachrichtigt in diesem Moment den Benutzer, welcher den *Merge* durchführen will und bieten die Chance diese Konflikte zu lösen. Es können beliebig viele Branches erstellt werden, welche schlussendlich aber immer mit dem sogenannten Master-Branch zusammengeführt werden. Diese Zusammenführung wird natürlich nur dann vorgenommen, wenn der Branch, welcher mit dem Master *gemerged* werden soll, absolut Fehlerfrei ist. Der Master-Branch sollte stets eine funktionierende Version des Softwareproduktes enthalten, weshalb niemals auf dem Master-Branch selbst entwickelt und getestet werden sollte.

### Automatisierter Build

Das Projekt muss stets aus der Summe seiner Teile vollautomatisch und von Grund auf neu gebaut werden können. Damit ist gemeint, dass zu keiner Zeit Fehlerhafter Quellcode vorliegen darf, welcher verhindern würde, einen Build zu erstellen. Fowler (2006) beschreibt hier zusätzlich, dass es Sinn macht Build-Werkzeuge zu nutzen, welche auf allen nötigen Plattformen zur Verfügung stehen, da Build-Werkzeuge von Entwicklungsumgebungen oftmals proprietäre Dateien beinhalten und damit nicht zwingend auf jedem System funktionieren werden.

### Selbsttestender Build

Während des Build-Prozesses muss das Produkt automatisch getestet werden. Es obliegt dem Entwickler diese Tests sinnvoll zu implementieren. Es ist wünschenswert, so viel Quellcode wie möglich durch Tests abzudecken. Dabei wird unterschieden zwischen:

sehr feingranularen Tests wie z.B. den *Unit-Tests*, welche nur sehr kleine, meist unabhängige Abschnitte Testen.

den mittelgroßen *Komponententests* die, wie der Name verrät, ganze Komponenten testen – also beispielsweise die Funktion mehrerer miteinander kommunizierender Klassen.

Den übergreifenden *Systemtests*, welche ein Produkt in seiner Ganzheit testen.

Tests spielen bei einer *continuous Integration* eine sehr zentrale, bedeutende Rolle, da nur durch ausreichende Testabdeckung die Funktionalitäten des Produkts bei jeder Integration geprüft werden können. Dadurch, dass mehrmals täglich integriert und gebaut wird, ist es schlicht unmöglich diese Tests manuell durchzuführen. Aus diesem Grund übernehmen diese Aufgabe die Tests, welche bei jeder Integration automatisch ausgeführt werden. Ein CI Server ist bei dieser Aufgabe hilfreich, weil dieser diese Tests automatisch anstoßen kann, sobald ein neuer Softwarestand integriert wurde. Auf diese Weise lässt sich das Produkt lückenlos, bei jeder Modifikation am *Code* bzw. bei jeder Neuerung von Grund auf erneut testen und ist zu jedem Zeitpunkt lauffähig.

### Häufige Integration

Einer der offensichtlichsten und wichtigsten Punkte dieser Auflistung ist die häufige Integration von Code. Die Entwickler sind dazu angehalten, so oft wie möglich, mindestens jedoch 1 Mal am Tag, Ihren Code in ein Versionskontrollsystem einzuchecken. Simon Wiest (2006: 34) spricht hier von einer kulturellen Veränderung, weil das häufige Einchecken des Quellcodes etwas ist, dass sich viele Entwickler erst angewöhnen müssen, bis es selbstverständlich wird. Bevor eingecheckt werden darf, muss natürlich die Version vorab lokal erfolgreich gebaut werden können um auf diese Weise zu vermeiden, dass Fehlerhafter Quellcode eingecheckt wird. Die Vorteile der häufigen Integration liegen auf der Hand:

Der aktuelle Entwicklungsstand und das aktuelle Master *Build* liegen nie weit auseinander, was enorm bei der Fehlerfindung hilft, da neue Fehler nur in dem neuen Delta zwischen den beiden Versionen auftreten verursacht werden können.

Die Entwickler können *Risikofreudiger entwickeln*, da sie jederzeit einen sogenannten *Roll-back* auf den letzten aktuellen Stand durchführen können.

Es kann viel *schneller auf Fehler reagiert werden*, weil diese durch das häufige Integrieren viel schneller erkannt werden (Wenn die Integration fehlschlägt)

Es liegt *zu jedem Zeitpunkt eines Projekts einen funktionierenden Stand* vor, auch wenn das nicht heißt, dass dieser fertig ist. Das liegt daran, dass durch ständiges integrieren und das anschließende Bauen des Softwarestandes stets gewährleistet ist, dass das Produkt nach dem bauen immer lauffähig ist. Ist das nicht der Fall, bzw. schlägt die Integration fehl, wird das Produkt entsprechend

Wiest (2010) erwähnt, abgesehen von diesen eher offensichtlichen Vorteilen, noch einen weiteren interessanten Effekt, den häufiges Integrieren mit sich bringt: eine höhere Motivation der Beteiligten. Er führt an, dass Entwickler welche oft integrieren, schneller Rückmeldungen ihrer Arbeit erhalten. Selbst negative Rückmeldungen sind seiner Meinung nach besser als keine Rückmeldungen, da Entwickler zumindest darüber informiert werden, dass etwas nicht funktioniert und sie sich direkt darum kümmern. (vgl. Wiest, 2010 : 28)

### Builds und Tests nach jeder Änderung

Ein absolut essenzieller Punkt einer CI ist, ebenso wie das häufige integrieren von neuem Code, das ebenso häufige bauen und testen des selbigen. Neuer Quellcode sollte bei einer guten CI in der Theorie nach jeder Änderung getestet und anschließend, bei erfolgreichen Tests, gebaut werden. Zu häufiges testen und bauen kann nach Wiest (2010: 38) aber zu einem Rückstau an Builds führen, bzw. zu Zeitintensiv ausfallen. Dr. Simon Wiest schlägt hier als groben Lösungsansatz vor, so oft wie es die Gegebenheiten zulassen zu bauen und nicht Zwingend nach jeder Änderung. Wie oft der Quellcode tatsächlich kompiliert und gebaut wird, hängt letztendlich von individuellen Faktoren ab und muss an diese angepasst werden.

Martin Fowler (2006) beschreibt, dass es Sinnvoll ist einen CI Server zu verwenden, welcher das Versionskontrollsystem überwacht. Wenn ein Projektmitglied einen neuen Stand zu dem verwalteten Repository[[3]](#footnote-3) hinzufügt, was auch *pushen* genannt wird, dann meldet das Verwaltungssystem dem CI Server das und dieses wiederum stößt den Build-Vorgang z.B. auf einem separaten Build-Server an und benachrichtigt das Mitglied über das Ergebnis. Durch die Verwendung eines CI Servers wird sichergestellt, dass nach *jedem* Commit ein Build erzeugt wird und die Entwickler zeitnah die Ergebnisse erhalten.



Abbildung : Ablauf der Kontinuierlichen Integration (CI), Quelle: Dr. Simon Wiest (2010: 14)

Abb. 1 veranschaulicht diesen Vorgang noch einmal bildlich. Auch wenn nicht zwingend ein CI-Server benötigt wird bietet es sich aus genannten Gründen an einen zu benutzen. Ein CI Server setzt allerdings wiederum schnelle Build-Zyklen voraus. Auch wenn ein CI Server von Grund auf neu entwickelt werden kann, gibt es bereits für alle erdenklichen Ansprüche ausgereifte Tools, welche in den allermeisten Fällen mehr als ausreichend sind (Paul M. Duvall et al, 2011: 85).

### Schnelle Build-Zyklen

Um die Vorteile einer CI voll auszuschöpfen ist, wie in 4.1.5 beschrieben, häufiges bauen essenziell. Es zeigt sich, dass es wichtig ist, die Build-Zyklen so kurz wie möglich zu halten um dadurch die Häufigkeit dieser erst möglich zu machen. Kent Beck (XXX) gibt einen Zeitaufwand von 10 Minuten für einen build als guten Richtwert an. In der Realität ist eine solche Zeit aber nicht immer ohne weiteres erreichbar und bedarf Anpassungen. Abgesehen von Aufrüstungen in Form von leistungsfähigeren Build-Servern, beschreibt Simon Wiest (2010: 39) drei Methoden um Build-Zeiten möglichst kurz zu halten.

1. Staffeln des Builds
   * Der Build selbst wird in mehrere Stufen bzw. kleinere Builds aufgeteilt. Fowler (2006) spricht von einem *Commit-Build* welcher als erstes gebaut wird. Das ist ein kurzer, nur schnelle Unit Tests beinhaltender Build, der zwar nicht komplett ist, dafür aber sehr schnell fertiggestellt werden kann. Dieser Build geht den nachgelagerten Builds voran und ist ausschlaggeben dafür, ob diese nachgelagerten Builds überhaupt erstellt werden oder nicht. Schlägt der *Commit-Build fehl*, werden nachgelagerte Builds meist ga nicht erst gebaut, das spart Zeit und Ressourcen. Außerdem bekommen Entwickler auf diese Weise schneller eine Rückmeldung (Wiest, 2006: 39)
2. Modularisierung
   * Das Zerlegen des Projekts in eigene, unabhängige Module bietet den Vorteil, dass nur diejenigen Module neu gebaut werden müssen, welche auch verändert wurden. Die übrigen Module lassen sich demnach ganz einfach aus den vorherigen Builds entnehmen und weiterverwenden (Wiest, 2006: 39).
3. Parallelisierung
   * Die Parallelisierung setzt eine Modularisierung voraus. Wenn Module nicht voneinander abhängig sind, spricht nichts dagegen diese parallel auf mehreren Rechnern zur selben Zeit zu bauen (Wiest 2006: 39).
   * Paul M. Duvall (et al, 2011: 96) empfiehlt das parallelisieren, oder auch *distributed integration* genannt als den letztmöglichen Versuch die Build-Dauer zu verkürzen, weil es ein äußerst komplexer Prozess ist.

### Tests in gespiegelter Produktionsumgebung

Auch wenn es der Idealfall wäre, ein Softwareprodukt stets in der Umgebung zu testen, in der es später tatsächlich eingesetzt wird, ist dies meist aus diversen Gründen nicht möglich. Zum einen spielen hier kosten eine Rolle, weil es schlicht unwirtschaftlich sein kann, extrem große Datenmengen zu transferieren oder aber extra dafür neue oder spezielle Hardware kaufen zu müssen. Zum anderen kommt ein Softwareprodukt in seinem Einsatzbereich nicht selten in Kontakt mit personenbezogenen Daten, welche aus juristischen Gründen nicht an Dritte weitergereicht werden dürfen. Das schließt auch Entwickler dieser Software mit ein, welche die Software nur im Auftrag entwickeln. Ein weiterer Grund kann sein, dass die Produktionsumgebung noch nicht existiert, weil sie sich selbst noch in Entwicklung befindet (Wiest, 2006: 40).

Auch wenn es durch diese und weitere Gründe nicht immer möglich sein wird, die Produktionsumgebung exakt abzubilden, sollte trotzdem versucht werden, eine Umgebung zu schaffen, welche der Originalen so nah wie möglich kommt. Dadurch können Fehler frühzeitig erkannt werden, welche unter Umständen nur in dieser einen speziellen Umgebung auftreten. Solche Fehler lassen sich demnach in einer schlechter abgebildeten Produktionsumgebung erst gar nicht auffinden bzw. reproduzieren und können im Nachhinein zu deutlich mehr Aufwand und Kosten führen, wie es die Abbildung einer genaueren Umgebung getan hätte.

Paul M. Duvall (et al, 2011: 194) rät dazu die Entwicklungsumgebung von Anfang an absolut sauber und in fein unterteilen Ebenen aufzubauen. Er empfiehlt folgende Ebenen nacheinander aufzubauen:

* Das Betriebssystem installieren
* Betriebssystem konfigurieren (z.B. Netzwerkeinstellungen, Benutzerkonten, Firewall-Einstellungen)
* Server Komponenten bereitstellen
* Server konfigurieren
* Software von Drittanbietern aufspielen
* Maßgeschneiderte Software installieren/Entwickeln (Das eigentliche Produkt)

Auf diese Weise lassen sich Fehlerquellen ausschließen, welche z.B. durch Fremdsoftware oder unbekannte Konfigurationen ausgelöst würden.

Simon Wiest (2006: 40) empfiehlt an dieser Stelle als etwas komfortableren Lösungsansatz *Cloud-Computing*[[4]](#footnote-4). Hierbei werden gewünschte Hardwarekonfigurationen über das Internet bereitgestellt und müssen somit nicht neu gekauft werden. Werden ganze Systeme bzw. Umgebungen über diesen Service bereitgestellt bekommen ist auch die Rede von *Platform as a Service* (PaaS). Für die Dauer der Nutzung des Services fallen selbstverständlich auch Kosten an. Da dieser Service allerdings nur für einen absehbaren Zeitraum in Anspruch genommen wird, liegen diese Kosten sehr wahrscheinlich unter den Kosten, welche andernfalls für die Neuanschaffung einer Nachbildung der gesamten Produktionsumgebung anfallen würden.

Letztendlich muss hier individuell entschieden werden, da es sich nicht pauschal beantworten lässt, welche der beiden Varianten die wirtschaftlichere ist.

### Einfacher Zugriff auf Build-Ergebnisse

Gerade die Verwendung einer Versionsverwaltung erleichtert den Zugriff auf sämtliche Softwarestände ungemein. Beteiligte an einem Projekt, seien es Entwickler, Teamleiter, Tester oder Kunden, bekommen mit Einführung eines CI-Systems (welches eine Versionsverwaltung einschließt) stets sämtliche Builds an einem zentralen Ort bereitgestellt. Dadurch wird laut Simon Wiest (2010: 40) die Wahrscheinlichkeit erhöht, das neue Softwarestände schneller zu Beteiligten gelangt, welche wiederum schneller Rückmeldung über die Qualität der jeweiligen Stände geben können.

### Automatisierte Berichte

Da ein CI-System stets den Überblick bzw. die Kontrolle über sämtliche Build-Prozesse hat lassen sich auf diese Weise sehr genau alle Schritte und Ergebnisse nachvollziehen. Diese Ergebnisse können gezielt an betroffene Entwickler gesendet werden. Dieser Ansatz wird auch *Aktive Benachrichtigung* genannt, da in diesem Fall das CI-System aktiv Nachrichten versendet und die Betroffenen Personen nicht selbst nach diesen Informationen suchen müssen. Diese Aktiven Benachrichtigungen sollten sehr detaillierte Informationen enthalten, welche z.B. die Auslöser eines Fehlers eingrenzen. Außerdem sollten Entwickler und andere Beteiligte nur bei wichtigen Ergebnissen benachrichtigt werden, um ein Stören dieser Funktion zu vermeiden (Wiest 2010: 41).

### Automatisierte Verteilung

Im letzten Schritt kann sich ein CI-System selbst noch um das verteilen eines Produktes kümmern. Es wird hierbei von *continuous deployment* gesprochen. Heutzutage gibt es Unternehmen, welche neue Softwarestände teils *mehrmals täglich* ausbringen. Paul M. Duvall (2011: 190) bezeichnet eBay, Amazon & Google als Vorzeigebeispiele in dieser Hinsicht.

Durch die bisherigen Schritte einer gut durchgeführten CI ist es möglich jederzeit funktionierende Software zu verteilen weil diese, wie es der Name verrät, kontinuierlich Integriert und bei jedem neuen Build getestet, und nur dann freigegeben wird, wenn sie sich Fehlerfrei bauen lässt.

Als besonders komfortabel ist an dieser Stelle die *Roll-back[[5]](#footnote-5)* Funktion hervorzuheben. Da ältere Softwarestände, wie schon mehrfach erwähnt, selbstverständlich archiviert werden, befinden sich Entwickler durch ein CI-System stets in der Lage die Ausbringung einer Neuen Software Version rückgängig zu machen, indem sie einfach einen *Roll-back* auf einen älteren Stand vollziehen. Von diesem älteren Stand ist bereits bekannt, dass er fehlerfrei läuft, da er logischerweise bereits im Einsatz war. Das führt wiederum dazu, dass Entwicklern mutiger agieren können, weil das CI System ihnen jederzeit ein virtuelles Fallnetz bietet.

## Nachteile der CI

Nachdem viele Vorteile einer gut durchgeführten CI im vorangegangenen Kapitel beleuchtet wurden soll an dieser Stelle auch auf die Nachteile, welche die Einführung einer CI mit sich bringen kann, eingegangen werden.

* Einige der beschriebenen Vorteile, wenn nicht sogar die meisten, werden nur dann zu bemerken sein, wenn die CI kompromisslos durchgeführt wird. Das hat zur Folge, dass *alle* Beteiligten Personen stets alle Praktiken bzw. Prinzipien, welche die CI fordert, einhalten. Diese Art zu Entwickeln wird für viele dieser Personen vermutlich eine Umstellung ihrer Arbeitsweise erfordern und wird sich nicht über Nacht einstellen. Diese Umstellung kann also relativ viel Zeit in Anspruch nehmen.
* Eine gemeinsame Codebasis bildet einen unverzichtbaren Teil der CI und setzt voraus, dass die Entwickler absolut gewissenhaft mit ihrem Code umgehen. Es sollte wie schon beschrieben nur Code in das VCS eingecheckt werden, welcher getestet wurde und stabil läuft. Es besteht die Gefahr, dass fehlerbehafteter Code durch unzureichende Testabdeckung nicht als solcher erkannt und in das VCS gelangt. Dadurch kann es im späteren Verlauf eines Projektes zu Fehlern kommen, welche sich nicht mehr so einfach finden lassen, da diese sich *nicht mehr* nur in dem kleinen Delta zwischen der vorangegangenen und der aktuellsten Version befinden können, sondern gegebenenfalls in Teilen von deutlich älteren Softwareständen verstecken.
* Ein CI-System wird nur so gut sein wie die Tests es zulassen. Da das CI-System nicht selbst den Sourcecode testet, sondern lediglich die Tests anstößt, welche die Entwickler zur Verfügung gestellt haben, liegt es letztendlich an diesen Tests und damit an den Entwicklern, wie gut eine CI funktioniert oder auch nicht. Diese Tests sollten, wie bereits in 4.1.3 erwähnt, so viel Sourcecode wie möglich abdecken. Aber wie beschrieben, sollte nicht nur der Sourcecode an sich, bzw. dessen Funktionen getestet werden, sondern ebenso sollte auf höheren Ebenen durch Komponenten- bzw. Systemtests ein Zusammenspiel dieser Fehler geprüft werden können. Diese Tests zu erstellen Bedarf wiederum viel Zeit.
* Da ein CI-System in den meisten Fällen zusätzliche Hardware in Form eines CI-Servers, eines Build-Servers und eventuell auch einem eigenen Server auf den ein Versionskontrollsystem läuft besteht, entstehen hier Initialkosten welche je nach Projektumfang variieren. Große Projekte, welche deutlich mehr Zeit benötigen um gebaut zu werden, setzen wahrscheinlich potentere Hardware bzw. sogar mehr Hardware (im Falle einer Parallelisierung des Build Prozesses) voraus als es kleinere würden.
* Da häufig Integriert wird, wird ebenso häufig gebaut, was Zeit in Anspruch nimmt. Auch wenn die Zeit, die ein Build benötigt um erstellt zu werden, wie in 4.1.6 erläutert, verkürzt werden kann, so kann es sehr Aufwendig sein diese Optimierungen vorzunehmen und letztendlich sind diese keine Garantie für die gewünschten Ergebnisse. Wenn die Build Zeiten aus welchen Gründen auch immer nicht kurz genug gehalten werden können, kann es wegen eines Rückstaus an Builds (Wiest 2010: 38) zu einer Verzögerung des gesamten Projektzeitplans kommen, weil Entwickler auf die Ergebnisse der Builds warten müssen.

Die überwiegende Mehrheit der Nachteile, welche die *Continuous Integration* verursachen kann, werden nur durch den Umgang mit selbiger ausgelöst – Also durch die Entwickler. Es gibt wie beschrieben auch solche, die nicht von den Entwicklern, bzw. den Beteiligten abhängen, wie zum Beispiel Initialkosten durch Neuanschaffung von Hardware, dennoch lässt sich sagen, dass die meisten Nachteile bei gewissenhafter und korrekter Durchführung nicht auftreten sollten. Diese korrekte Durchführung muss natürlich erst einmal zur Routine werden, was Zeit und Erfahrung bedarf. Trotzdem überwiegen die Vorteile der CI klar deren Nachteilen, vor allem dann, wenn die Nachteile durch eine korrekt angewendete CI erst gar nicht auftreten.

# Stand der Technik

Dieses Kapitel beleuchtet die Diagnosetoolkette von Daimler und erläutert deren Funktion. Außerdem wird darauf eingegangen wie die Mitarbeiter mit der Toolkette arbeiten und wie sie diese weiterentwickeln. Alle Beschreibungen der Toolkette beziehen sich auf deren aktuellen Stand (10.03.2018). Es ist wichtig zu verstehen, wie genau mit der Toolkette gearbeitet wird um im Anschluss daran das Konzept der *Continuous Integration* theoretisch auf diese anwenden zu können. In Kapitel 3.1 und 3.2 wurde bereits auf die Vor- und Nachteile der *Continuous Integration* eingegangen. Diese sollten auch hinsichtlich einer möglichen Anwendung der CI auf die Diagnosetoolkette gegeneinander abgewogen werden. Nur wenn nach gründlicher Betrachtung die Vorteile überwiegen, macht es Sinn das Konzept der CI auch praktisch auf die Diagnosetollkette anzuwenden.

## Ralf

## Michl

## Nuding

# Die Theoretische Anwendung der Continuous Integration auf die Diagnosetoolkette von Daimler

# Eingeschlagener Realisierungsweg

Dieses Kapitel beschäftigt sich mit dem praktischen Teil der Bachelor Arbeit. Das übergeordnete Ziel war das Tool *DTS Monaco*, welches innerhalb der Diagnosetoolkette zum Einsatz kommt, automatisiert zu testen und dem Anwender die Testergebnisse, in visuell und strukturell aufbereiteter Form, zur Verfügung zu stellen. Die feiner definierten Ziele sind in Kapitel 2 „Ziele“ zu nachzulesen.

## Verwendete Tools, Softwaretechnologien und Datenstrukturen

In den folgenden Unterkapiteln wird genauer auf die Tools & Softwaretechnologien eingegangen, welche zur Erstellung des Programms benutzt wurden.

### Python als Programmiersprache

Der gesamte Source-Code wurde in mithilfe der Programmiersprache *Python* geschrieben. Python ist eine interpretierte, objektorientierte Programmiersprache (Python.org, 2018). Eine Interpretierte Programmiersprache hat den Vorteil, dass sie nicht erst durch einen *Compiler*[[6]](#footnote-6)in eine für die jeweilige Plattform ausführbare Datei übersetzt werden muss, sondern direkt von einem *Interpreter*[[7]](#footnote-7)analysiert und ausgeführt werden kann. Es sei gesagt, dass dafür zwar das eigentliche Ausführen des Quellcodes länger dauert als das Ausführen von bereits kompiliertem Code, allerdings ist der Quellcode eines in Python verfassten Programms dafür meistens recht kurz. Dadurch wird Zeit während der Entwicklung gespart. Eine Interpretierte Programmiersprache bringt allerdings auch die Gefahr mit sich, dass zur Laufzeit Fehler auftreten können, welche beim Entwickeln mit einer anderen, *Compiler-basierten* Programmiersprachen, wie z.B. *C/C++* spätestens beim Kompilieren aufgefallen wären. Der Interpreter unterstützt den Entwickler allerdings ausreichend bei der Fehlersuche, indem er eine *Exception*[[8]](#footnote-8) anzeigt und die Programmausführung an dieser Stelle abbricht*.* Das grenzt den Bereich, in dem es zu einem Fehler gekommen ist, stark ein und vereinfacht die Suche nach diesem, sodass der Nachteil weniger stark ins Gewicht fällt als angenommen werden könnte.

Mit Python lassen sich im Vergleich zu anderen Programmiersprachen deutlich schneller Ergebnisse erzielen. Einer der Gründe warum das so ist, liegt in der sehr einfach gehaltenen Syntax des Codes. Statt, wie es zum Beispiel in Java der Fall ist, einzelne Code-Blöcke in geschweiften Klammern zu schreiben, geschieht diese Form der „Trennung“ in Python durch einfaches Einrücken des Codes, welcher sich in einem Block befindet. Statt das Ende einer Zeile z.B. mit einem Semikolon zu versehen, übernimmt diese Funktion in Python ein einfacher Zeilenumbruch. Zugegeben sind das nur Feinheiten, allerdings summieren sich diese bei mehreren tausend Zeilen Code schnell zu einer beträchtlichen Summe an gesparter Zeit zusammen. Des Weiteren kommt Python mit einer sehr reichhaltigen und mächtigen *Library*[[9]](#footnote-9)daher, die das Entwickeln von neuem Quellcode oft stark vereinfacht und vorantreibt. Gerade diese reichhaltige Bibliothek an Funktionen hat bei der Entwicklung des Projekts enorm geholfen.

Es gibt mittlerweile einige unterschiedliche Python Distributionen, welche sich vor allem in der nativ vorhandenen Auswahl an vorinstallierten Modulen bzw. ihrer Library unterscheiden. Während diesem Projekt wurde ausschließlich mit der Python Distribution *Anaconda* entwickelt auf welche im Folgenden eingegangen wird.

### Die Python Distribution *Anaconda*

*Anaconda* ist eine relative junge Python Distribution und war früher unter dem Namen „Continuum Analytics“ bekannt. Eine Python Distribution enthält neben dem eigentlichen Interpreter meist auch nützliche P*ackages[[10]](#footnote-10)* und Werkzeuge oder sogar Entwicklungsumgebungen um mit der Programmiersprache zu arbeiten. Wie schon erwähnt beinhaltet *Anaconda* eine große Library welche über 100 Packages beinhaltet, wobei es die Möglichkeit gibt unzählige weiter zu installieren. An dieser Stelle kommt der Package Manager *Conda* ins Spiel mit welchem dies äußerst komfortabel gelöst wird. Mit *Conda* lassen sich bequem weiteren Packages suchen und installieren. Diese Packages kommen aus dem Hauseigenen Repository *repo.continuum.io*, was den Vorteil mit sich bringt, dass Anaconda® diese Packages selbst überwacht bzw. wartet und auf dem aktuellsten Stand hält (Conda.io, 2017). Somit ist sichergestellt, dass keine veralteten Packages in ein Projekt gelangen, welche unter Umständen Schwierigkeiten verursachen.

Es sei gesagt, dass es kein *Anaconda* braucht, um Packages zu installieren oder um generell mit Python zu arbeiten. Dennoch bietet eine Distribution wie *Anaconda* dem Entwickler Vorteile - meist in Form von Komfortfunktionen - welche den Arbeitsalltag und damit das Entwickeln mit Python erheblich erleichtern können.

### PyCharm als Entwicklungsumgebung

Auch wenn es theoretisch möglich ist, ohne eine Entwicklungsumgebung bzw. eine Integrierte Entwicklungsumgebung (engl.: I*ntegrated Development Environment, kurz: IDE*)zu entwickeln, macht es ohne jeden Zweifel Sinn eine zu verwenden. Ohne hierbei zu tief ins Detail zu geben, bietet eine IDE dem Entwickler eine grafische Benutzeroberfläche, welche den Umgang mit ihr äußerst einfach gestaltet. Eine IDE bietet stets Zugriff auf alle Informationen über den gesamten Projektumfang und eine Syntaxprüfung, welche beim Entwickeln unglaublich hilfreich ist. Eine Syntaxprüfung zeigt dem Entwickler sofort an, wenn z.B. eine Funktion falsch geschrieben wurde, was wiederum beim Ausführen zu einem Fehler und damit einem Programmabsturz (auch *Crash* genannt) führen würde. Auch lässt sich über eine IDE gegebenenfalls eine Versionsverwaltung realisieren, welche in dem Fall dieser praktischen Arbeit allerdings nicht von *PyCharm* selbst übernommen wurde. Außer einer Syntaxprüfung hilft ein integrierter *Debugger* beim Auffinden von Fehlern. Dieser ermöglicht z.B. das setzen von Haltepunkten an welchen die Programmausführung angehalten wird. Das bietet dem Entwickler beispielsweise die Möglichkeit die Werte von verschiedenen Variablen zu genau diesem Zeitpunkt zu überprüfen, um damit zu überprüfen ob diese Werte den Erwartungen entsprechen oder auch nicht. Das kann äußerst hilfreich beim Auffinden von Fehlern sein. Außerdem lässt sich in einer IDE der geschriebene Code direkt ausführen und muss nicht erst über die Kommandozeile gestartet oder sogar erstmal kompiliert werden.

In diesem Projekt wurde ausschließlich mit *PyCharm* entwickelt. Innerhalb des Teams in dem das Projekt entstand, wurde bereits damit gearbeitet, weshalb die Entscheidung nicht schwer viel. PyCharm bietet eine übersichtliche und moderne Benutzeroberfläche. Außerdem ist eine Code-Vervollständigung integriert, welche das Programmieren um ein Vielfaches komfortabler macht. Diese Code-Vervollständigung schlägt dem Entwickler Beispiele vor, welche er an dieser Stelle benutzen könnte. Außerdem funktionieren *Anaconda* und dessen Package-Manager *Conda* im Zusammenspiel mit *PyCharm* sehr gut. Innerhalb von *PyCharm* wird der Benutzer einen *Interpreter* auswählen müssen, welcher in diesem Fall dann *Anaconda* ist. Die Benutzeroberfläche von *PyCharm* bietet dem Benutzer dann die Möglichkeit Über *Conda* ganz bequem neue Packages zu suchen und zu installieren.

### *Git* im Zusammenspiel mit *TortoiseGit* und *GitLab* als Versionskontrollsystem

Eine Versionskontrolle ist vor allem in Hinblick auf eine gut durchgeführte *Continuous Integration* enorm wichtig. Aber auch wenn eine CI nicht oberste Priorität haben sollte, bietet es sich an ein VCS zu benutzen, da die Vorteile der Versionierung von Quellcode sehr vielfältig und dank bereits vorhandener Lösungen auch einfach zu handhaben sind.

In Kapitel 4.1.1 wurde bereits auf Versionskontrollsysteme eingegangen und deren Grundlegenden Funktionen erläutert. Die Basis des Verwendeten VCS war in diesem Projekt das Verteilte Versionskontrollsystem (im folgenden als DVCS bezeichnet) *Git.*



Abbildung : Visualisierung eines Verteilten Versionskontrollsystems, Quelle: Git (19. Oktober 2017): Seite „1.1 Getting Started - About Version Control“.

Bei einem DVCS gibt es, wie es Abb. 2 veranschaulicht, im Grunde genommen keine Trennung zwischen Entwickler- und Serverumgebung, was einen Server im Grunde genommen nicht mehr nötig macht (René Preißel, Bjørn Stachmann, 2017: 2). Dennoch wird in der Regel einer benutzt, was besonders der Strukturierung eines Projektes zu Gute kommt. René Preißel und Bjørn Stachmann (2017: 3) geben als Beispiel, welche *für* die Nutzung eines Servers spricht, spezifische Repositorys an, welche sinnvollerweise auf einem Server liegen sollten. Da wäre zum Beispiel das sogenannte *Blessed Repository*, aus welchem die Fertigen Softwarestände erstellt werden. In diesem Repository liegt also zu jedem Zeitpunkt ein lauffähiges Produkt. Zum anderen nennen René Preißel und Bjørn Stachmann (2017: 3) an dieser Stelle das *Shared Repository*, welches als Austausch-Repository dient. Ob man einen Server für spezielle Repositorys benutzt und welche das dann sind, ist individuell zu entscheiden. Es lässt sich aber sagen, dass ein Server das Risiko von Datenverlusten und damit auch finanziellen Verlusten minimieren kann.

Ein Verteiltes Versionskontrollsystem wie *Git* bietet die Möglichkeit *Branches* zu erstellen, um paralleles Arbeiten so bequem wie möglich zu gestalten. Dadurch kann sehr *flexibel* zwischen verschiedenen Aufgaben gewechselt werden. Entwickler können für verschiedene Aufgaben unterschiedliche *Branches* erstellen. Diese können im späteren Verlauf wieder *gemerged* (also zusammengeführt) werden ohne Gefahr zu laufen, dass unabhängige Teilaufgaben, welche zur selben Zeit in Bearbeitung sind, vermischt werden. *Git* lässt die Entwickler fast alle Operationen lokal durchführen. Das heißt es können *Branches* erstellt, oder *commits* ausgeführt werden, ohne eine Internetverbindung zu haben. Damit lässt sich also *Offline* die Versionierung ebenso realisieren wie *Online*. Zu einem späteren Zeitpunkt, an dem eine aktive Internetverbindung vorhanden ist, lassen sich *offline* getätigte Operationen nachträglich *einchecken*, d.h. zum Beispiel zum *Blessed Repository* hinzufügen. Da die meisten Operationen offline durchgeführt werden können, ist auch nur selten eine Kommunikation mit dem Server oder anderen Arbeitsrechnern notwendig, was die Performance steigert. (René Preißel und Bjørn Stachmann, 2017: 3)

Um einfach mit der Versionsverwaltung zu arbeiten, empfiehlt es sich einen Dienst wie *GitLab* in Anspruch zu nehmen. *GitLab* ist eine Webanwendung, welche den Umgang mit einem Versionskontrollsystem sehr intuitiv gestaltet. Die Webanwendung bietet den Nutzern im Hinblick auf das Projektmanagement erhebliche Vorteile. *Roadmaps* (dt.: Produktpläne) können einfach erstellt werden, um Projekte in Phasen aufzuteilen und damit das Zeitmanagement kontrollierbarer zu machen. Eine *Issue-Tracking* (dt.: Fehlerverfolgungs-) Funktion, unterstützt Entwickler dabei, den Überblick über vorhandene Probleme zu behalten, was gerade bei großen Projekten sehr von Vorteil ist. Außerdem können *Milestones* (dt. Meilensteine) erstellt werden, was ebenso wie eine *Roadmap* positivzum Zeitmanagement beitragen kann. *GitLab* bietet mit *GitLab CI/CD* außerdem die Möglichkeit eine CI zu realisieren. Von dieser Funktion wurde in diesem Projekt allerdings nicht Gebrauch gemacht. Für einen späteren Zeitpunkt wäre das aber definitiv eine Option.

Um Softwarestände zu *committen* und in das jeweilige *Repository einzuchecken*, wurde das Programm *TortoiseGit* benutzt. Dieses Programm vereinfacht das arbeiten mit einem VCS indem es eine grafische Benutzeroberfläche bereitstellt, um mit dem VCS zu kommunizieren. Statt einen *Commit* über die Kommandozeile auszuführen, reicht mithilfe von *TortoiseGit* ein Rechtsklick auf den Projektordner und über den Menüpunkt „*TortoiseGit“* werden alle verfügbaren Funktionen angezeigt.



Abbildung : TortoiseGit Kontext Menü

In Abb.3 ist das Kontextmenü von *TortoiseGit* zu sehen. Es lässt sich daran erkennen, dass es deutlich angenehmer ist mit dieser Oberfläche zu arbeiten, als die einzelnen Befehle über die Kommandozeile auszuführen. Außer dem zusätzlichen Komfort, bietet *TortoiseGit* stets nur die Funktionen an, welche auf dem jeweils angeklickten Objekt auch Sinn ergeben. Außerdem bietet das Programm eine *Diff-Funktion* welche beim Auflösen von *Merge-Konflikten* hilfreich ist. Diese Funktion lässt den Anwender synchron durch zwei unterschiedliche Versionen derselben Datei suchen und markiert entsprechende Unterschiede bzw. Konflikte, die der Anwender dann beseitigen kann.

Wie auch *GitLab* ist *TortoiseGit* sicherlich nicht notwendig für eine gute Versionskontrolle, allerdings gibt es keine konkreten Nachteile (Die Einarbeitungszeit außen vorgelassen). Der Komfort und die Effizienz, sowie das Projektmanagement können von diesen *Tools* nur profitieren, weshalb sie in diesem Projekt gerne und häufig genutzt wurden.

### Testing mit Unit Tests

Das *Testing* eines Software-Produktes ist generell enorm wichtig und gerade im Hinblick auf die CI unverzichtbar. Wie bereits erwähnt, kann eine CI nur dann effektiv Nutzen bringen, wenn ausreichend Tests den Code bei jeder Integration und jedem *Build-*Vorgang überprüfen.

Die zukünftige Vision, die Diagnosetoolkette von Daimler mit CI weiter zu entwickeln, setzt also ein hohes Maß an Testbarkeit voraus. Dazu gehören dann Ebenso kleinere Tests, wie z.B. *Unit-Tests,* als auch umfangreichere Tests wie Komponenten- und Systemtests (Siehe dazu Kapitel 4.1.3). Das gesamte Projekt stellt aus Sicht einer *Continuous Integration der Diagnosetoolkette* einen umfangreichen Komponententest dar, welcher die Komponente bzw. das *Tool* „DTS Monaco“ testet. Der *Code* dieses Komponententests, welcher wie beschrieben das Projekt selbst darstellt, muss natürlich ebenfalls getestet werden. Das geschieht durch die feineren *Unit-Tests*, die kleinere Teile einer Komponente testen, wie beispielsweise einzelne Funktionen. Wenn eine dieser Funktionen einen *Input* in Form von Dateien, Werten oder sonstigem erwartet, wird dieser *Input* dem *Unit-Test* in der Regel durch sogenannte *Dummy-Daten*[[11]](#footnote-11) übergeben. Mit diesen Dummy-Daten wird die Funktion ausgeführt und erzeugt eventuell einen *Output*, welcher letztendlich mit den erwarteten Ergebnissen abgeglichen werden kann. Wenn der *Output* der Funktion mit den Erwartungen übereinstimmt, gilt der Test als erfolgreich und umgekehrt als fehlgeschlagen.

Funktionen die keinen *Output* erzeugen können trotzdem ausreichend geprüft werden indem innerhalb der Funktion beispielsweise Variablen auf deren aktuellen Wert, Typen geprüft werden. Werden hier andere Werte ausgelesen, als angenommen wurde, kann eine *Exception* geworfen werden, welche den Test mit einer individuellen Fehlermeldung abbricht. Wenn es zu keiner *Exception* kommt, dann ist der Test soweit es die Überprüfung durch *Exceptions* betrifft Fehlerfrei durchlaufen worden. Daran ist zu erkennen, dass es einzig in den Händen des Entwicklers liegt, die Tests so umfangreich wie möglich bzw. wie nötig zu gestalten. Die *Exceptions*, also erwartete Ausnahmen, müssen von diesem gut überlegt und implementiert werden. Es ist allerdings nicht unbedingt von Vorteil, *jedes* noch so kleine Detail zu testen, weil diese Zeit benötigen um durchlaufen zu werden. Es sollte also ein Mittelmaß an Testabdeckung, Testgenauigkeit und *Performance* gefunden werden, welches sich nicht Pauschal im Voraus bestimmen lässt.

### GUI automatisierung mit pywinauto

*Pywinauto* ist eine Sammlung von Modulen, welche dazu benutzt werden kann *Microsoft-Windows*-Benutzeroberflächen automatisiert zu bedienen. Dabei wird entweder zuerst eine Applikation gestartet und mit einer sogenannten *Application-*Instanz verknüpft, oder aber eine bereits laufende Applikation mit dieser *Application*-Instanz verbunden.

Anschließend ist es möglich, über eben diese *Application*-Instanz auf die jeweils verknüpft Anwendung zuzugreifen. Es lassen sich sämtliche Kind-Elemente der Anwendung als Objekte geben. Diese Kind-Elemente sind z.B. *Buttons*, Textfelder oder andere Bedienelemente einer Benutzeroberfläche. Diese Elemente lassen sich im Folgenden dann einfach programmatisch bedienen, indem in etwa ein Klick auf einen *Button* simuliert wird, was über die *click()*-Funktion möglich ist.

### Luigi

*Luigi* ist ein Python *Package* das zur Erstellung von komplexen *Pipelines Jobs*[[12]](#footnote-12) benutzt werden kann. Dabei unterstützt es den Entwickler bei dem *Workflow Management*[[13]](#footnote-13), der Visualisierung, der Fehlerbehandlung und der Auflösung von Abhängigkeiten zwischen den einzelnen Aufgaben, was auch als *Dependency Resolution* bezeichnet wird (vgl. Python Software Foundation, 2017). Das Projekt selbst umfasst eine Vielzahl von kleineren Aufgaben bzw. Funktionen, welche nur im Zusammenspiel in der Lage sind das *Tool „DTS Monaco“* zu testen. Dieses Zusammenspiel wird mithilfe von *Luigi* realisiert.

Die einzelnen Aufgaben, welche von *Luigi* koordiniert werden, sind jeweils in eigenen Klassen abgebildet die von der *luigi.Task* Klasse erben müssen und werden *Tasks* genannt. Diese Vererbung setzt die Implementierung von 3 Funktionen zwingend voraus. Diese und weitere Eigenschaften eines *Tasks* werden mit dem folgenden *Pseudo-Code* Ausschnitt beispielhaft erläutert:

Import luigi

class Luigi\_Example\_Task(luigi.Task):

[1] parameter = luigi.Parameter(default = 0)

[2] **def requires**(self):

return Another\_Task()

[2] **def run**(self):

foo = 3

bar = 2

if for > bar:

\_out = self.output().open('w')

\_out.write('Teststring')

\_out.close()

[3] **def output**(self):

return luigi.LocalTarget('/tmp/foo/text.txt')

[1] Jeder *Task* kann eigene Parameter besitzen und mit diesen aufgerufen werden. Die gezeigte Beispiel-Klasse kann also mit Luigi\_Example\_Task(parameter=2)aufgerufen werden, wobei in diesem konkreten Beispiel der *Task* als Parameter parameter der Wert '2' beim Aufruf mitgegeben wird. Innerhalb des *Tasks* kann dann dieser Parameter beliebig verwendet werden, was zum Beispiel bei der Parallelisierung von *Tasks* unverzichtbar ist.

[2] Die erste der zu implementieren Funktionen ist die requires() Funktion. Durch diese wird festgelegt, welche *Tasks* bereits erfolgreich durchlaufen sein müssen, bevor die aktuelle gestartet wird. In diesem Beispiel muss also der Pseudo *Task* Another\_Task abgeschlossen sein. Allerdings kann selbstverständlich auch eine Vielzahl an Aufgaben vorausgesetzt werden, anstatt nur einer. Es ist ebenfalls möglich dieselbe *Task* mehrfach aufzurufen, um so parallel ablaufende *Pipelines* zu ermöglichen. Diese Parallelisierung wurde innerhalb des Projektes realisiert. Auf die Art und Weise, wie das gemacht wurde, wird in Kapitel 4.2 eingegangen. Die requires() Funktion ist also der Teil von *luigi,* welcher für die *Dependency Resolution* zuständig ist. (vgl. readthedocs, 2015 „Tasks“)

[3] Die zweite und vermutlich interessanteste Funktion welche implementiert werden muss ist die run()Funktion. In dieser wird der eigentliche *Code* des *Tasks* ausgeführt. Außerdem wird in der run() Methodeder Inhalt des Outputs festgelegt, also in diesem Beispiel der Text „Teststring“. Es muss zwar nicht zwingend etwas Sinnvolles in die *Output-*Datei geschrieben werden, es kann aber durchaus helfen z.B. Informationen über den Ablauf des jeweiligen *Tasks* in diese *Output*-Datei zu schreiben. Das kann bei der späteren Analyse des *Workflows* helfen. (vgl. readthedocs, 2015 „Tasks“)

[4] Letztendlich setzt die Vererbung noch die Implementierung der output() Methode voraus. Auch wenn diese unscheinbar ist, so ist sie enorm wichtig, denn ein *Task* gilt in *luigi* nur dann als abgeschlossen, wenn er einen *Output* erzeugt hat. Grundsätzlich wird in der output() Methode festgelegt, wohin die *Output*-Datei geschrieben wird. (vgl. readthedocs, 2015 „Tasks“)

Mithilfe von *luigi* und der *Tasks* lassen sich so äußerst komplexe *Pipelines* aufbauen, wobei durch die einfach zu handhabende *Dependency Resolution*, welche wie beschrieben durch die requires() Methode realisiert wird, der Ablauf des *Workflows* absolut genau vorgegeben werden kann. In parallel verlaufenden Abschnitten der mit *luigi* erstellten *Pipeline* werden Verästelungen, welche zu Fehlern führen, abgebrochen, sodass die *Tasks* der nächsten parallel verlaufenden Zweige ausgeführt werden. Dies lässt sich am einfachsten an folgendem Schaubild erkennen, welches den *Dependency-Graph* zeigt den *luigi* mit sich bringt.



Abbildung : Dependency Graph von Luigi

Der *Dependency Graph* veranschaulich optimal in welchem Bezug die einzelnen *Tasks* zueinander Stehen und von welchen anderen *Tasks* sie abhängen. Der Graph ist von unten nach oben zu betrachten. Der unterste *Task* markiert den Startpunkt, von dem die restlichen ausgehen. Die grauen Linien repräsentieren deren Abhängigkeiten. Abgesehen davon, lässt sich in dieser Ansicht auf einen Blick nachvollziehen, welche *Tasks* bereits erfolgreich durchlaufen wurden (grün, „*Done*“), welche gerade ablaufen (blau, „*Running“)*, welche noch ablaufen werden (gelb, „*Pending*“) und welche fehlgeschlagen sind (rot, „*Failed*“). Wird der Mauszeiger über den jeweiligen Punkt bewegt, werden weitere Informationen über den Status angezeigt. Möglich macht die Visualisierung der *Central Scheduler*.

Dieser *hostet* die gesamte Visualisierung aller *Tasks* lokal, sodass sie im *Browser* unter dem *localhost* mit Port Nummer 8082 (*localhost:8082*) einsehbar sind. Diese Visualisierung lässt sich selbstverständlich auch *online hosten* was bei Projekten mit vielen Teammitgliedern von Vorteil ist, weil es jedem Mitglied stets möglich ist den aktuellen Stand der *Pipeline-Jobs*, zu überprüfen. Der *Central Scheduler* stellt außerdem sicher, dass nicht zwei Instanzen desselben Tasks ablaufen (vgl. readthedocs, 2016 „Using the Central Scheduler“).

*Luigi* ist zwar ebenso wie viele andere Module, welche bei der Arbeit an dem Projekt benutzt wurden nicht zwingend notwendig, dennoch hat es dessen Qualität gerade durch die Visualisierung enorm erhöht. Die *Dependency Resolution* ist außerdem sehr komfortabel gelöst und wäre ohne *luigi* deutlich umständlicher zu realisieren gewesen. Nach einer relativ kurzen Einarbeitungszeit lassen sich so mit *luigi* schnell erste *Workflows* erstellen und koordinieren.

### DTS Monaco

### Wichtige Datenstrukturen

#### smr-d Dateien

<X>

#### ODX Dateien

ODX ist die Abkürzung für *Open Diagnostic Data Exchange* und ist ein Dateiformat, das auf der Auszeichnungssprache XML (*Extensible Markup Language*) basiert. In ODX-Dateien werden alle Informationen hierarchisch abgespeichert, welche bei der Fahrzeugdiagnose relevant sind. Dieses Dateiformat dient ebenso als Schnittstelle zwischen Fahrzeugherstellen und deren Zulieferern. ODX Dateien und deren Aufbau sind beiden Beteiligten bekannt, sodass eine Zusammenarbeit erleichtert wird. Es gibt verschiedene Kategorien von ODX-Daten, welche durch verschiedene Endungen zu erkennen sind. Jede dieser Kategorien ist für eine andere Art von Daten gedacht, wobei hier und im Folgenden stets das ODX-D Format gemeint ist, wenn nicht explizit ein anders vermerkt wird. In diesen ODX-D Dateien werden Diagnosedaten gespeichert. (vgl. Vector, Seite „Lösungen für ODX“, 2018)

Im Verlauf des Projekts spielen ODX Daten eine wichtige Rolle, denn *DTS Monaco* wird durch das Durchlaufen von Simulationen und das Erstellen von Kurztests auf dessen Funktionsweise getestet. Um diese Simulationen bzw. die Kurztests durchzuführen, benötigt das *Tool* verschiedene Daten von den jeweiligen Steuergeräten, mit welchen dies geschehen soll. Ein Teil genau dieser Daten sind in den ODX Dateien der jeweiligen Steuergeräte zu finden.

#### Das Dateiformat YAML

YAML Dateien sind äußerst hilfreich beim Abspeichern von Daten. YAML selbst ist ebenso wie XML eine Auszeichnungssprache, mit der sich hierarchische Strukturen speichern lassen. Beim Entwickeln mit *Python* sind YAML-Dateien deshalb so geschickt, weil beispielsweise *Dictionarys*[[14]](#footnote-14) sehr unkompliziert in diesem Dateiformat abgespeichert und umgekehrt ebenso einfach Informationen aus diesen herausgezogen werden können. Das Projekt selbst bzw. dessen Funktionen bedienen sich während dem Programmablauf häufig dieser Dateien. Anders als z.B. XML-Dateien sind YAML-Dateien für Menschen sehr übersichtlich lesbar, was der folgende kleine Ausschnitt einer in *Python* geschriebenen Verschachtelung eines *Dictionarys* in einer Liste zeigt:



Abbildung : Skizzenhaftes Beispiel einer YAML-Datei

## Das automatisierte Testen des Diagnosetools DTS Monaco

Um zu verstehen wie das Programm welches den praktischen Teil der vorliegenden Arbeit darstellt funktioniert, ist es am sinnvollsten den Programmauflauf von Beginn bis zum Ende zu beleuchten. Da dieser Programmablauf durch die in Kapitel 4.1.6. *Tasks* von *Luigi* koordiniert wird, werden die folgenden Kapitel auf die einzelnen *Tasks* in chronologischer Reihenfolge eingehen. Zuvor wird allerdings erst einmal darauf eingegangen, wie *DTS Monaco* getestet wird, um einen Überblick über die Funktionsweise des automatisierten Tests zu gewähren.

Wie bereits erwähnt wird mit dem Programm das Diagnosetool *DTS Monaco* automatisiert getestet. Dazu werden Steuergerätedaten aus ODX- und SMR-Dateien benutzt, mit denen mithilfe des *Tools* *DTS Monaco* Simulationen durchgeführt und Kurztests erstellt werden. Die Simulationen werden im besten Fall an das jeweilige Steuergerät bzw. dessen Diagnosedaten angepasst, sodass auf Steuergerät-spezifische Dienste getestet werden kann. Angepasst wird die Simulation mithilfe einer Simulationsdatei, welche mit dem XML Format aufgebaut ist. In dieser Simulationsdatei werden die Diagnosedienste festgelegt, welche getestet werden sollen. Ein Beispiel, wie die Abfrage eines solchen Diagnosedienstes aussieht, bietet der Nachstehende Ausschnitt der Simulationsdatei:

<DiagService pdupattern=“false“>

<Request>22 F1 00</Request>

<Response>62 F1 00 02 5B 08 03</Response>

</DiagService>

Die Form einer Abfrage eines Dienstes ist immer gleich. Das <DiagService>-*Tag* markiert den Anfang. Der Inhalt des folgenden <Request>-*Tags* beschreibt den Diagnose Dienst, der abgefragt werden soll. Zuletzt wird innerhalb des <Response>-*Tags* die zu erwartende Antwort festgelegt. Die ersten 3 *Bytes* sind dieselben wie die der Anfrage (*Request*), lediglich um 40 erhöht und markieren den Beginn der Antwort (*Response*). Die *Bytes* nach diesen ersten 3 *Bytes* sind für den *Payload* gedacht, also die eigentliche Antwort auf die gestellte Anfrage. In der Simulationsdatei wird die Anfrage eines Diagnose Dienstes und dessen erwartete Antwort (bei fehlerfreier Funktion) also genau festgeschrieben. Aus Zeitgründen wurde auf die dynamische Anpassung der Simulationsdatei an die jeweiligen Steuergeräte (ECUs) verzichtet, sodass lediglich diejenigen dienste getestet werden, zu welchen alle Steuergeräte in der Lage sind. Ist eine Simulation vollständig durchlaufen, so liegen am Ende die Ergebnisse eines Kurztests vor, welche im Anschluss überprüft werden können. Die Ergebnisse der Kurztests werden in XML Format abgespeichert. Für die Überprüfung der Ergebnisse wird wiederrum die Simulationsdatei benötigt, denn wie bereits beschrieben sind die erwarteten Antworten aller zu testenden Diagnosedienste bereits vorab in dieser Simulationsdatei festgeschrieben worden. Es werden also die Ergebnisse des Kurztests mit den erwarteten Antworten verglichen. Eine Simulation gilt daher sinnvollerweise als Fehlerfrei, wenn die Erwartungen mit den Ergebnissen übereinstimmen. Während des Programmablaufs wird dieser Vorgang für jedes Steuergerät wiederholt und die Ergebnisse der Simulationen werden gut strukturiert innerhalb des Projektordners abgelegt, sodass zu einem späteren Zeitpunkt einfach auf sie zugegriffen werden kann.

### Das Erstellen der Basis-Verzeichnisse

Sämtliche mit den Tests in Verbindung stehenden Daten müssen für den Benutzer im Nachhinein leicht auffindbar sein, weshalb es wichtig war eine logische Ordnerstruktur für diese Daten zu schaffen, welche intuitiv durchsucht werden kann. Die ersten 3 *Tasks* sind genau dafür zuständig. Durch die CreateWorkingDirectory()-*Task* wird das Hauptverzeichnis für den Programm-*Output* erstellt. Der Name dieses Verzeichnisses und dessen Pfad werden, wie alle anderen Verzeichnis-Pfade und -Namen innerhalb der *Config*-Datei festgelegt, welche stets im *Source-*Verzeichnis liegen muss. Bevor das Hauptverzeichnis (im Folgenden als *Working-Directory* bezeichnet) erstellt wird, wird geprüft ob es bereits existiert oder nicht. Es wird selbstverständlich nur dann erstellt, wenn dies nicht der Fall ist. Als nächstes wird innerhalb von CreateLuigiDirectory()ein Ordner für den *Output* der *Luigi*-*Tasks* erstellt, welcher innerhalb des *Working-Directorys* liegt. Die Pfade sind relativ zum *Working-Directory* angegeben, weshalb gewährleistet wird, dass diese sich entsprechend anpassen. Als letztes wird durch die CreateImportantFilesDirectory()-*Task* ein Ordner für absolut essenzielle Dateien erstellt, welche später zwingend gebraucht werden. Jeder *Task* erzeugt wie in Kapitel 4.1.6 beschrieben einen *Output,* in welchem beispielsweise Informationen über den Verlauf der *Task* gespeichert werden können. Allerdings wurde die Speicherung wichtiger, *Task*-relevanter Informationen auf einem anderen Wege realisiert. Es war wichtig, dass diese Daten möglichst zusammengefasst in wenigen Dateien zu finden sind. Statt also für jeden *Task* den *Output* zu speichern, welcher danach logischerweise in einer jeweils separaten Datei liegt, wurden diese wichtigen Daten in YAML-Dateien gespeichert und zusammengefasst. In diesen sind zu jedem *Task* der Name, die Zeit und der Status gespeichert, wie im Folgenden zu sehen:

<X>

Um zu entscheiden, ob ein *Task* erfolgreich abgeschlossen ist oder nicht, muss eine jeweils individuelle Prüfung dieser *Task* stattfinden, was durch unkomplizierte keine Abfragen geschieht. Im Falle der Erstellung der Ordner, wird beispielsweiße kurz *nach* der eigentlichen Erstellung abgefragt, ob dieser Ordner mittlerweile existiert oder nicht. In der im YAML-Formt aufgebauten Infodatei ist später schnell nachzuvollziehen, welche *Tasks* erfolgreich waren und welche nicht. Diese Informationen sind über die *Luigi-Web-Visualisierung* zwar genauso, bzw. sogar ansprechender aufbereitet, allerdings nur solange der *Central-Scheduler* (vgl. Kapitel 4.1.6) noch läuft bzw. die Webansicht verfügbar ist. Wenn dies jedoch nicht mehr der Fall ist, so sind diese Informationen immer noch in den eben beschriebenen Infodateien verfügbar.

### Das Einloggen in das Diagnoseportal

Im Diagnoseportal werden alle Daten gelagert, welche relevant für die Fahrzeugdiagnose sind. Diese Plattform ist selbstverständlich Passwortgeschützt und nur befugten Mitarbeitern zugänglich, weshalb sich mit den eigenen *Login-*Daten zuerst einmal auf das Portal geschaltet werden muss. Da der *Login*-Name für das Portal von der aktiven *Windows-Session* übernommen wird, kann dieser bequem über os.getLogin()erhalten werden. Das os-Modul beinhaltet viele Standardfunktionen, welche im Zusammenhang mit dem Betriebssystem stehen. Das Passwort muss der Nutzer vorab in der *Config*-Datei (im Folgenden nur noch *Config* genannt) abspeichern, sodass es automatisch ausgelesen werden kann. Das geschieht in der parse\_data() Methode der Klasse Config. Da die *Config* selbst im YAML-Format abgelegt ist, lassen sich alle Daten dieser Datei schnell über die load()Funktion des *ruamel\_yaml*-Moduls auslesen. Auf diese Weise werden während des gesamten Ablaufs des Programms oft Daten aus YAML-Dateien gelesen oder in welche geschrieben, wobei in letzterem Fall statt der load()- wird die dump()-Funktion benutzt wird, welche ebenfalls im *ruamel\_yaml*-Modul zur Verfügung steht. Um nicht bei jedem Lesen bzw. Schreiben einer solchen Datei die Routine neu zu implementieren, sind diese in load\_yaml() bzw. save\_yaml() ausgelagert.

Nachdem nun also Login-Name sowie Passwort vorhanden sind, kann über einen sogenannten *subprocess* ein Befehl über die Kommandozeile ausgeführt werden. Das Diagnoseportal hat glücklicherweise eine Kommandozeilen-Schnittstelle, die ausführlich dokumentiert ist, sodass es möglich ist alle vom Diagnoseportal (im Folgenden mit „DP“ abgekürzt) benötigten Daten über diese Schnittstelle zu beziehen. Der *subprocess* wird also aufgerufen und bekommt mitgeteilt, dass er die *diagConCmd.exe* mitsamt dem Konsolenbefehl, welcher den *Login* startet, und den *Login*-Daten als Startparameter ausführen soll:

subprocess.call(executable='pfad/zur/diagConCmd.exe', args=[

'start',

'-login',

'-user', os.getLogin(),

'-password', passwort\_aus\_der\_config\_datei])

Der *Login*-Versuch gibt als Rückgabewert den Wert 0 zurück, wenn er erfolgreich war, sodass anhand dieses Wertes einfach festgelegt werden kann, ob die aufrufende *Task* erfolgreich war oder nicht.

### Das Downloaden und parsen der Metaview

Nachdem der *Login* in das DP erfolgreich verlief, lassen sich alle dort gelagerten Dateien herunterladen bzw. diejenigen für die der Nutzer/Mitarbeiter freigeschaltet ist. Um wissen zu können welche ECU-Daten überhaupt nötig sind, muss zunächst einmal die *Metaview* heruntergeladen werden. In ihr sind die Namen aller Steuergeräte im XML-Format gespeichert und nach Fahrzeugtyp (PKW, Van, etc), Baureihe und Derivat sortiert. Der *Download* wird genau wie der *Login* über einen *subprocess* bewerkstelligt, welcher sich der *diagConCmd.exe* bedient. Die *Metaview* wird direkt in den Ordner, welcher in dem in 4.2.1 beschriebenen Task CreateImportantFilesDirectory() erstellt wurde.

Sobald der *Download* erfolgreich abgeschlossen wurde, können die Namen aller Steuergeräte, sowie deren Fahrzeugtyp, Baureihe und Derivat aus der *Metaview* bezogen werden (Im Folgenden wird bei der Umwandlung bzw. dem herausfiltern von Daten von *parsen* gesprochen). Dieser Vorgang wird von dem *Task* ParseMetaView() angestoßen. In diesem *Task* wird der Funktion parse\_file()die Metaview übergeben, welche diese über das *etree*-Modul in ein E*lementTree*-Element umwandelt, wodurch das gesamte XML-Dokument als Baumstruktur abgespeichert wird. Das hat den Vorteil, dass sich dieses Element anschließend mit den Werkzeugen, welche das *etree*-Modul mit sich bringt, durchsuchen und bearbeiten lässt. Dieses Element wird rekursiv durchlaufen und entsprechende Informationen werden ausgelesen und währenddessen in ein *Dictionary* gespeichert. Bei diesem Vorgang wird sichergestellt, dass jedes ECU nur ein einziges Mal im *Dictionary* vorkommt.

Sobald das Dictionary vollständig erstellt ist und damit alle verfügbaren ECU Namen beinhaltet, müssen diese lediglich noch in eine flache Liste gespeichert werden, die in den folgenden *Tasks* benutzt werden kann. Dies wird von dem *Task* GetEcuNames() erledigt. Diese ruft die Funktion get\_all\_ecu\_names() der Klasse DiagConParser auf, welche ganz einfach alle ECU-Namen aus dem kurz zuvor erstellten *Dictionary* der Metaview ausließt. Da jeder ECU-Name nur ein einziges Mal vorhanden ist und stets als *Key* des *Dictionarys* in derselben, obersten Ebene vorliegt, können diese ganz einfach über die keys() Funktion ausgegeben und in eine Liste gespeichert werden. Bevor diese Liste letzten Endes noch in eine YAML-Datei geschrieben wird, wird überprüft ob sie überhaupt Einträge beinhaltet. Ist das der Fall, wird die YAML-Datei erstellt und der *Task* kann als abgeschlossen betrachtet werden.

### Das Erstellen der ECU Verzeichnisse innerhalb des Working Directorys

Im nächsten Schritt muss für jedes ECU ein eigener Ordner erstellt werden, in dem Simulationsrelevante Daten, deren Ergebnisse sowie alle anderen mit dem Steuergerät in Verbindung stehenden Informationen abspeichern zu können. Das war eine wichtige Zielvorgabe, weil Ergebnisse des automatisierten Tests von *DTS Monaco* am Ende in einer logischen, leicht verständlichen Struktur vorliegen sollten um eine spätere Analyse einfach möglich zu machen.

Der *Task* CreateDirectories() ruft die create\_directories()-Methode der BasicFunctions()-Klasse auf, welche genau diese Aufgabe übernimmt. Dieser Methode wird die im vorherigen Schritt erstelle Liste mit allen ECU-Namen übergeben, sowie der Pfad in dem die Verzeichnisse erstellt werden sollen. Anschließend wird über diese Liste iteriert und für jeden Eintrag (welcher wie bereits mehrfach erwähnt einen ECU-Namen repräsentiert) ein gleichnamiger Ordner erstellt, wenn dieser nicht bereits existiert. Im selben Zug werden noch weitere Verzeichnisse und Dateien in diesen Ordnern erstellt, welche später mehr Übersicht ermöglichen. Diese Verzeichnisse und Dateien sind:

1. Ein „*Report*“-Ordner
   * In diesem werden später Ergebnisse der Kurztests, welche mit der Simulation in *DTS Monaco* erstellt werden, abgelegt
2. Ein Ordner für die PDX Datei
   * PDX-Dateien sind Archivdateien, die eine Vielzahl an Steuergeräte-Diagnosedaten enthalten, von denen einige im späteren Verlauf gebraucht werden. Weil die Übersicht darunter leiden würde, werden diese PDX-Dateien nicht direkt im ECU-Verzeichnis entpackt, sondern ganz einfach in dem PDX-Ordner.
3. Eine Info-Datei im YAML-Format
   * In dieser Info-Datei werden alle direkt mit dem Steuergerät in verbindung stehenden *Task*-Informationen (Name, Zeitpunkt, Status) abgespeichert. Der Aufbau dieser Datei ist auf der Abbildung aus Kapitel 4.2.1 zu sehen.
4. Eine Datei für alle notwendigen Diagnosedaten aus den entsprechenden ODX-Dateien
   * Auf ODX-Dateien wurde in Kapitel 4.1.7.2 bereits kurz eingegangen. Wichtig zu wissen ist, dass diese Dateien Daten enthalten, welche für die Durchführung einer Simulation bzw. der Erstellung eines Kurztests zwingend notwendig sind.

Nachdem alle Verzeichnisse und Dateien vollständig erstellt wurden, ist der Aufbau der gesamten Ordnerstruktur and dieser Stelle abgeschlossen.

### Download der Diagnosedateien aus dem Diagnoseportal

Die bisher beschriebenen *Tasks* laufen jeweils nur genau ein einziges Mal zu Beginn der Automatisierung ab, weil sie unabhängig von Steuergeräte-spezifischen Daten ablaufen. Ab dem *Download* von Diagnosedaten aus dem Diagnoseportal ist dies allerdings nicht mehr der Fall, denn diese Daten sind logischerweise für jedes Steuergerät anders, sodass der Workflow ab diesem *Task* in viele Äste parallel aufgeteilt wird, was in Abbildung <X> aus Kapitel 4.1.6 schon zu sehen war. Den *Workflow* ab diesem Zeitpunkt parallel aufzubauen statt die Tasks weiterhin sequenziell ablaufen zu lassen, dient nicht nur dazu, dass der *Dependency Graph* übersichtlicher wird. Es wird dadurch auch möglich, dass parallel verlaufende Äste des *Workflows* nicht voneinander abhängen. Wenn also einer dieser Äste aufgrund eines fehlgeschlagenen *Tasks* abgebrochen wird, hat das nicht zur Folge, dass die übrigen Äste ebenfalls abgebrochen werden. Da diese wie beschrieben *keine* gegenseitigen Beziehungen besitzen, läuft jeder Ast unabhängig vom Ergebnis der anderen ab.

Diese Parallelisierung wird durch die dynamische Vergabe von *Dependencys* in *Luigi* erreicht. In Kapitel 4.1.6 wurde kurz angedeutet, dass genau für diesen Zweck die Parameter eines *Tasks* wichtig sind. Ebenfalls wurde erwähnt, dass ein *Task* von einer Vielzahl an vorherigen *Tasks* abhängen kann. Der nachstehende *Code-*Ausschnitt zeigt die requires()-Methode eines *Tasks,* der von vielen identischen *Tasks* abhängt, welche allerdings alle in einem eigenen Kontext ausgeführt werden, was durch die Vergabe von unterschiedlichen Parameterwerten geschieht.

Task A(luigi.Task):

Def requires(self):

ecu\_liste = ['ecu1', 'ecu2 ']

Task\_B\_liste = [Task\_B(ecu) for ecu in ecu\_liste]

Return Task\_B\_liste

In dem *Code*-Ausschnitt wird beispielhaft ein Task\_A gezeigt, welcher von einer Anzahl an Task B’s abhängig ist. Die ecu\_liste enthält ECU-Namen, welche bei der Iteration durch die Liste als Parameter an Task\_B weitergegeben werden. Zuletzt wird mit der Zeile return Task\_B\_liste angegeben, dass Task\_A von allen in Task\_B\_liste befindlichen *Tasks* abhängt, welche in diesem Beispiel 2 Einträge umfasst : Task\_B('ecu1') und Task\_B('ecu2'). Das bedeutet, dass bevor Task\_A ablaufen kann zuerst die beiden Instanzen von Task\_B abgeschlossen sein müssen. Diese beiden Instanzen sind eindeutig durch ihre Parameter identifizierbar, weshalb diese zwingend einzigartige Werte haben müssen.

Genau auf diese Weise wird nun für jedes Steuergerät der *Task* DownloadFiles()aufgerufen, wobei jeder *Task* hierbei als Parameter den eindeutigen ECU-Namen erhält. Der ECU-Name ist nicht nur eindeutig und deshalb geeignet dafür als eine *ID* für den jeweiligen *Task* zu fungieren, sondern er ist auch noch bei der Kommunikation mit dem Diagnoseportal hilfreich, um festzulegen von welchem Steuergerät Daten heruntergeladen werden sollen. Für die Simulation in *DTS Monaco* werden zum einen Daten aus der ODX-Datei des Steuergerätes und zum anderen dessen SMR-D-Datei benötigt. Die ODX-Datei ist in einer PDX-Datei verpackt/archiviert, weshalb es nötig ist diese herunterzuladen. Beide dieser Dateien (ODX/PDX) werden, ebenfalls über einen *subprocess* mithilfe der *DiagConCmd.exe* aus dem Diagnoseportal heruntergeladen. Damit die richtigen Dateien heruntergeladen werden, muss der *DiagConCmd.exe* der genaue Name der Dateien übergeben werden. Dieser wird aus der sogenannten *DocumentVersionList geparsed*. Das ist ein weiteres XML Dokument, welches kurz vor dem *Download* der ODX- bzw. PDX-Dateien aus dem Diagnoseportal heruntergeladen werden muss. In diesem Dokument stehen die benötigten Informationen, also die Namen der ODX- und PDX-Datei des jeweiligen Steuergerätes sowie deren Versionen. Als *Download*-Pfad wird für die PDX-Datei, weil diese im nächsten Schritt noch entpackt werden muss, das extra angelegte PDX-Verzeichnis verwendet wohingegen die SMR-D-Datei direkt in das ECU-Verzeichnis abgelegt werden kann. Wie schon beim *Download* der *Metaview*, liefert die *DiagConCmd.exe* auch dieses Mal einen Rückgabewert anhand dessen ausgemacht werden kann, ob der *Download* funktioniert hat oder nicht.

### DTS Monaco automatisiert Starten

Sobald *alle* für einen Kurztest benötigten Dateien über das Diagnoseportal heruntergeladen wurden (Die jeweilige PDX- & SMR-D-Datei), kann das zu testende *Tool DTS Monaco* gestartet werden. Ab diesem Zeitpunkt wird die Oberfläche dieses *Tools* (die GUI) automatisiert bedient. Das wird über die *Python* Modulsammlung *pywinauto* bewerkstelligt, welche dem Entwickler Werkzeuge bietet, *Microsoft Windows* Benutzeroberflächen programmatisch zu bedienen (siehe Kapitel 4.1.6).

Der *Task* OpenMonaco() ruft die Funktion startMonaco() der Klasse MonacoAutomation() auf. Diese prüft im ersten Schritt, ob die Anwendung *DTS Monaco* bereits läuft oder nicht, indem sie mithilfe der Funktion check\_element() der Hilfsklasse MonacoAutomationHelper() nach einer laufenden Applikation mit dem Titel „*DTS Monaco [Daimler] – No workspace loaded*“ sucht. Sollte diese Hilfsfunktion eine entsprechende Anwendung finden, bedeutet das, dass DTS Monaco bereits läuft und dementsprechend nicht noch einmal gestartet werden muss. Dafür sucht die Funktion maximal 20 Sekunden lang immer wieder nach der gewünschten Anwendung bzw. deren Fenster und macht nach jeden Versuch 1 Sekunde Pause. Findet diese Funktion in diesem Fall das gesuchte Fenster jedoch nicht, veranlasst sie mithilfe von openMonaco() sofort den Start von *DTS Monaco*. Diese Funktion stellt wiederum ein Objekt zur Verfügung, über das auf die Anwendung zugegriffen werden kann und prüft zum Schluss mit genau derselben Funktion (check\_element()) auch noch ob die Anwendung tatsächlich läuft.

Da DTS Monaco dem Benutzer sofort nach dem Start ein Warnungs-Dialogfenster anzeigt, welches stets bestätigt werden muss um das Tool zu starten, wird diese Bestätigung im Zuge des Startvorgangs ebenso von der openMonaco() Funktion übernommen. Wie bereits beschrieben, lassen sich Kind-Elemente einer Anwendung bzw. eines Anwendungs-Fensters suchen und ausgeben. Dafür muss zuerst das *Window*-Element der entsprechenden Anwendung beschafft werden. Dabei muss von einer *Application*-Instanz die *window()*-Funktion aufgerufen werden:

Warning\_dialog\_window = application.window(title=“Warning-Title“)

Anschließend bietet in dem gezeigten Beispiel die Variable Warning\_dialog\_window einen Zugriffspunkt auf den Warnungs-Dialog.

Soll wie in diesem Fall ein spezieller *Button* „geklickt“ werden, so wird dieser erst einmal über die child()-Funktion gesucht, wobei wieder das passende *title*-Attribut als Parameter mitgegeben werden muss. Der *title* ist in diesem Fall der *Button*-Text („*Yes, I am*“).

Zuletzt kann dieser Button durch den Aufruf der click()-Funktion „geklickt“ werden, wodurch der Warnhinweis programmatisch bestätigt wird.

### Die PDX-Datei Entpacken

Nachdem die PDX-Datei bereits heruntergeladen wurde, muss sie anschließend natürlich entpackt werden, weil es sich dabei um ein komprimiertes Archiv handelt. Der *Task* UnzipPDX() nimmt sich dieser Aufgabe an, wobei er nach dem Entpacken auch noch die richtige ODX-Datei aus den entpackten Dateien sucht und diese in den dafür vorgesehenen Ordner schiebt. Zu Beginn wird erst überprüft ob in dem PDX-Ordner eine Datei liegt, die entpackt werden kann. Über das *Python-*Modul os lässt sich durch die listdir(path)-Funktion eine Liste ausgeben, welche alle Namen der Dateien enthält die unter dem angegebenen Pfad zu finden sind. In dem vorliegenden Fall *muss* diese Liste genau *einen* Eintrag enthalten, welcher die PDX-Datei repräsentiert. Ist dies der Fall, so wird die Funktion UnzipPDX() ausgeführt, welcher als Übergabeparameter den Pfad zur PDX-Datei, sowie den Pfad des jeweiligen PDX-Ordners, in welchen das Archiv entpackt werden soll, übergeben bekommt.

Entpackt wird die Datei dann schließlich mithilfe des *Python*-Moduls zipfile bzw. dessen Funktion ZipFile(). Nachdem das geschehen ist, liegen in dem Angegebenen Pfad eine Vielzahl an Dateien, wovon nur eine tatsächlich nötig ist: die *richtige* ODX-Datei. Es gibt allerdings unter Umständen einige ODX-Dateien nach dem entpacken, weshalb die passende erst mit Sicherheit bestimmt werden muss. Dafür wird über alle Dateien, welche vorhanden sind, iteriert und nur solche die mit „*.odx-d*“ enden, werden berücksichtigt. Durch das etree-Modul wird aus der jeweiligen ODX-Datei ein „*Baum“* gemacht*,* welcher anschließend mithilfe von *xpath-*Ausdrücken durchsucht werden kann. Es wird dann nach einem ganz bestimmten Knoten in diesem Baum gesucht, welcher nur in der richtigen ODX-Datei zu finden ist, wodurch diese mit Bestimmtheit gefunden werden und in das dafür vorgesehene Verzeichnis geschoben werden kann. Letzteres wird durch die shutil.move(Datei, ZielVerzeichnis)realisiert.

### Informationen aus der ODX-Datei herausziehen

Im vorangegangenen Schritt wurde die für die Simulation notwendige ODX-Datei bereitgestellt. Diese beinhaltet Informationen, welche für die Durchführung einer Simulation unerlässlich sind: z.B. die Anfrage- und Antwort-*IDs* des *CAN-Bus-Systems*[[15]](#footnote-15). In der bereits beschriebenen Simulations-Datei werden diese Informationen für das jeweils zu simulierende Steuergerät eingetragen, sodass eine Simulation bzw. ein Kurztest durchgeführt werden kann. Stimmen die eingetragenen Daten nicht, so kommt es zu Fehlern während der Simulation, weil bspw. Andere Anfrage- oder Antwort-*IDs* erwartet wurden. Diese *IDs* sind für jedes Steuergerät unterschiedlich, da alle ECUs innerhalb eines *CAN-Bus-Systems* auf demselben Kommunikationsnetz laufen. Sie müssen sich also eindeutig identifizieren lassen, was über diese *IDs* geschieht. Stimmt also eine dieser *IDs* nicht, so wird das jeweilige Steuergerät nicht als solches erkannt und ein Kurztest würde nur noch Fehler produzieren.

Um also diese Informationen aus der ODX-Datei zu *parsen,* führt der *Task* ParseODX() zuerst die Funktion load\_file() aus, welche die jeweilige ODX-Datei abermals mithilfe des etree-Moduls in eine Baumstruktur umwandelt. Anschließend werden durch die beiden Funktionen parseBaseVariants() und parseVariants()alle benötigten Daten und viele weitere aus der Baumstruktur herausgezogen und in ein *Python-Dictionary* gespeichert. Nachdem die Baumstruktur durchlaufen und aus deren Knoten die benötigten Informationen alle in dieses *Dictionary* gespeichert wurden, wird dieses letztendlich noch in eine YAML-Datei geschrieben, welche in dem jeweiligen ECU-Verzeichnis abgelegt wird.

### Die Simulationsdatei mit den richtigen Daten Befüllen

Wie bereits beschrieben müssen in der Simulationsdatei passende Informationen für das Steuergerät, für welches eine Simulation ausgeführt werden soll, eingetragen werden. Diese Daten befinden sich dank des letzten *Tasks* allesamt in der *ODX\_data.yml* des jeweiligen ECU-Verzeichnisses.

Wieder einmal wird mithilfe des etree-Moduls aus der Simulationsdatei, welche innerhalb des *DTS Monaco-*Verzeichnisses einen festen Platz hat, eine Objekt mit Baumstruktur erzeugt. Der Inhalt der *ODX\_data.yml-*Datei wird durch die load\_yaml() Hilfsfunktion, welche bereits in Kapitel 4.2.2 kurz erläutert wurde, wieder in ein *Dictionary* umgewandelt. Anschließend werden alle nötigen Daten aus diesem *Dictionary* gelesen und in Variablen geschrieben. Der Wert dieser Variablen wird dann in die entsprechenden Stellen bzw. Knoten des Baumes der Simulationsdatei geschrieben. Dazu wird das Objekt selbst mithilfe der find()-Funktion durchsucht und dem gewünschten Knoten der Wert zugewiesen:

Sim\_Baum.find('Pfad/zum/Knoten ').text = Variablenwert

Die find()-Funktion wird mit dem etree-Modul zur Verfügung gestellt. *Etree* Elemente (*etree.Element()*) können mit dieser Funktion sehr einfach durchsucht werden indem der Funktion bei Aufruf der Pfad des gesuchten Knotens mitgegeben wird.

### Die SMR-D-Datei in das dbr-Verzeichnis von DTS Monaco kopieren

Der *Task* CopySmrdFileToDbrDirectory() ist sehr simpel, dennoch unerlässlich. Durch ihn wird die SMR-D-Datei des Steuergerätes in das dbr-Verzeichnis von *DTS Monaco* geschoben. In diesem Verzeichnis sucht *DTS Monaco* während der Simulation nach SMR-D-Dateien. Damit eine Simulation überhaupt ablaufen kann muss in diesem Ordner also die passende SMR-D-Datei liegen. Die Funktion move\_smrd\_to\_dbr\_dir()wird aufgerufen, bekommt den Pfad der richtigen SMR-D-Datei übergeben und durchsucht dann zuerst einmal den dbr-Ordner. Dort wird die, von der letzten Simulation vorhandene, SMR-D-Datei gelöscht und die aktuelle mithilfe von shutil.copy(smrd\_pfad, ziel\_pfad) hereinkopiert.

### Start der Simulation

Nachdem nun die Simulations-Datei mit den richtigen Daten befüllt wurde, die SMR-D-Datei im dbr-Verzeichnis von *DTS Monaco* liegt und die Anwendung bereits geöffnet wurde, kann jetzt die eigentliche Simulation gestartet werden.

Dazu ruft der *Task* StartSimulation() die Funktion startSimulation() der Klasse MonacoAutomation() auf und übergibt ihr den Pfad unter dem der *Report* (Also das Ergebnis des Kurztests) gespeichert werden soll. Die Funktion StartSimulation() ruft sequenziell weitere Funktionen auf welche zusammengenommen die die gewünschte Simulation in *DTS Monaco* ausführen. Zuerst wird durch OpenWorkspace() und selectWorkspace() der *Workspace* geöffnet – Es wird stets derselbe *Workspace* genutzt, dieser muss lediglich einmal händisch in dem *Workspace*-Verzeichnis von *DTS Monaco* erstellt werden.

Die Funktion runSimulation() startet dann die Simulation selbst, anschließend werden mithilfe von save\_finas\_report() und save\_report() die Ergebnisse in Form von Reports, welche im XML-Format gespeichert werden, in den dafür vorgesehenen Verzeichnissen gespeichert.

Die Funktionsweise der genannten Funktionen basiert auf dem Prinzip, welches in Kapitel 4.2.6 erläutert wurde. Es sei an dieser Stelle jedoch erwähnt, dass jede Funktion zu Beginn stets mit der bereits beschriebenen Hilfsfunktion check check\_element() überprüft, ob das Anwendungs-Fenster mit dem sie arbeiten muss bereits existiert. Das ist wichtig, denn gerade in Anwendungen mit einer komplexen Benutzeroberfläche kann es unter Umständen einige Zeit dauern, bis diese fertig aufgebaut sind. Wird bspw. versucht einen „Klick“ auf einen *Button* zu einem Zeitpunkt zu simulieren, an dem es diesen noch nicht gibt, führt das zu einem *Crash* und das Programm wird beendet. Deshalb ist es wichtig auf die entsprechenden Oberflächen zu warten. Feste Wartezeiten eignen sich hier nur bedingt, weil es nicht immer möglich ist diese Zeit vorher zu bestimmen. Aus diesem Grund wurde der genannte Lösungsansatz gewählt. Da die Zeitspanne, in der die Funktion check\_element() nach einem Anwendungsfenster sucht, in eben dieser festgelegt werden kann, ist es unproblematisch diese im Nachhinein anzupassen. Als Standard wurden hier 20 Sekunden gewählt, was sich zum jetzigen Zeitpunkt als absolut ausreichend herausgestellt hat.

### Schließen des DTS Monaco Workspaces

Nachdem die Simulation erfolgreich abgelaufen ist und die *Reports* erstellt und anschließend in den richtigen Verzeichnissen abgelegt wurden, muss der *Workspace* in DTS Monaco noch geschlossen werden. Wie alle anderen *Tasks* welche die GUI automatisiert steuern behilft sich auch der *Task* CloseWorkspace() an den Funktionen des Moduls pywinauto. Nachdem wiedermal mittels child\_window() der *Button* zum schließen des *Workspaces* gesucht und programmatisch geklickt wird, muss anschließend noch auf gleich Weise ein Dialogfenster mit einem simulierten Klick auf den „Nein“-*Button* geschlossen werden, sodass der *Workspace* erfolgreich geschlossen werden kann.

### Der Vergleich der Kurztestergebnisse mit den Vorgaben der Simulation

Zu diesem Zeitpunkt liegen die Ergebnisse des durchgeführten Kurztests vor. Diese müssen nun noch mit den erwarteten Ergebnissen verglichen werden die, wie bereits in Kapitel 4.2.8 sowie 4.2.9 beschrieben, in der Simulationsdatei im *DTS Monaco*-Verzeichnis zu finden sind. In Kapitel 4.2 wurde der Inhalt dieser Simulationsdatei beispielhaft anhand eines einzigen Diagnosedienstes (*Diagnoseservice*) gezeigt. Diese Diagnosedienste beinhalten wie bereits beschrieben die Anfrage- sowie Antwort-Kennung sowie die erwartete Antwort selbst, in Form von Hexadezimalzahlen.

Während der Simulation in *DTS Monaco* werden diese Dienste simuliert und deren Ergebnisse in den *Reports* der Kurztests festgehalten. Wenn ein Diagnosedienst also erfolgreich simuliert wurde, so wird die aus Hexadezimalzahlen bestehende Antwort des Steuergerätes auf eben diesen Dienst genau die selbe sein wie die erwartete Antwort, welche in der Simulationsdatei für diesen Diagnosedienst hinterlegt war. Es ist also notwendig diese Ergebnisse mit der den Einträgen der Simulationsdatei abzugleichen, um festzustellen, ob alle während der Simulation angefragten Diagnosedienste erfolgreich waren oder nicht.

Da die *Reports* (also die Ergebnisse) der Kurztests in einem XML-Dokument gespeichert werden (nachzulesen in Kapitel 4.2 bzw. 4.2.11), müssen die benötigten Informationen aus diesen *Reports* zuerst herausgefiltert werden. Das erledigt die Funktion parseXML() der Klasse XMLParser(). Hierbei wird der *Report* zu Beginn mithilfe des etree-Moduls in ein Objekt mit Baumstruktur (etree.Element()) umgewandelt, weil dieses Objekt leichter durchsucht werden kann. Nachdem also das etree.Element() vorliegt, können mithilfe der Funktion find(Knotenname) schnell einzelne Knoten des Elements gefunden werden . Diese Knoten werden innerhalb der Funktion wiederum auf die gleiche Art nach Kind-Knoten abgesucht, wodurch immer tiefer die Baumstruktur eingedrungen wird, bis letztendlich der gewünschte Knoten erreicht ist. Dieser besitzt abermals Kind-Knoten, über die jetzt iteriert werden kann. Gleichzeitig wird deren Inhalt in ein *Dictionary* gespeichert, sodass diese *Dictionary* nach dem Ablauf der Schleife sämtliche benötigten Ergebnisse des *Reports* enthält.

Die Simulationsdatei enthält bisher, wie bereits erwähnt, nur eine feste Menge an Diagnosediensten, welche alle Steuergeräte bedienen können. In Zukunft sollen diese Dienste aus der Simulationsdatei *geparsed* und in ein Dictionary gespeichert werden, welches exakt den gleichen Aufbau hat, wie das *Dictionary* in dem die *Report*-Ergebnisse gespeichert wurden. Weil geplant ist, die Simulation an das jeweilige Steuergerät anzupassen, muss dieser Vorgang infolgedessen nach jedem Neuaufbau der Simulations-Datei durchgeführt werden, damit zu jeder neuen Simulations-Datei jeweils ein *Diactionary* vorhanden ist, welches schnell und einfach mit den *Report*-Ergebnissen verglichen werden kann.

Bisher ist dieses „Simulations-*Dictionary*“ unter dem Namen „sim\_data.yml“ im Hauptverzeichnis des Projekts zu finden und wird aus den genannten Gründen nach jeder Simulation zum Abgleich mit deren Ergebnissen benutzt. Weil die beiden *Dictionarys* absolut identisch aufgebaut sind, reicht die folgende *Pseudocode*-Zeile um zu überprüfen, ob auch identische Werte enthalten sind:

Check = (Simulations\_dictionary == report\_output\_dictionary)

Check ist True wenn die beiden *Dictionarys* gleich sind und andernfalls logischerweise False.

### Das Bündeln aller Tasks und das anschließende Starten des gesamten Testvorgangs

Nachdem in den Kapiteln 4.2.1 bis 4.2.13 alle *Tasks* und deren Funktionen erläutert wurden, wird im Folgenden noch erklärt, wie genau diese *Tasks* aufgerufen werden. Wie bereits erwähnt, werden die Abhängigkeiten zwischen den einzelnen *Tasks* über deren requires()-Methode bestimmt. Ebenso wurde erwähnt, dass ein *Task* von mehr als nur einem weiteren *Task* abhängen kann, sondern von einer beliebigen Anzahl. Genau dieser Möglichkeit wird sich beim Aufruf der *Tasks* bedient. Ein sogenannter *Wrapper-Task* (luigi.WrapperTask()) fasst, wie der Name andeutet, sämtliche *Tasks* in der richtigen Reihenfolge zusammen, sodass letztendlich nur noch dieser eine *Wrapper-Task* aufgerufen werden muss. Die ersten *Tasks*, welche ohne Parallelisierung, also nur ein einziges Mal ablaufen, werden in eine *Python* Liste zusammengefasst. Die nachfolgenden Tasks, also diejenigen die in direktem Bezug zu einem speziellen Steuergerät stehen, werden wie in Kapitel 4.2.5 veranschaulicht, mithilfe der jeweiligen Steuergerätenamen als luigi.Parameter gebündelt. Dafür wird über sämtliche Namen aller Steuergeräte iteriert und bei jedem Durchlauf eine Instanz des jeweiligen *Tasks* zusammen mit dem aktuellen Steuergerätenamen als Parameter erstellt. Diese Instanzen werden alle in eine Liste gespeichert. Dieser Vorgang wird für alle *Tasks*, welche Steuergerätespezifisch sind, durchgeführt, sodass am Ende für jeden dieser *Tasks* eine Liste mit einer gewissen Anzahl an Instanzen des selbigen verfügbar ist.

Alle während dieser Prozesse entstandenen Listen werden wiederum übersichtshalber in eine einzige Liste gebündelt. Es liegt zu diesem Zeitpunkt also eine einzige *Python* Liste vor, welche *alle Tasks* die ausgeführt werden sollen, beinhaltet. Diese Liste wird dem Wrapper-Task mit dem Namen PipelineTasks() in der requires()-Methode zugewiesen, womit dieser Abhängig von allen anderen Tasks wird. Wenn PipelineTasks() aufgerufen wird, müssen folglich alle anderen Tasks davor ausgeführt werden.

Bevor dieser Wrapper-Task allerdings aufgerufen und damit das Programm gestartet werden kann, müssen noch 3 kleinere Vorbereitungen getroffen werden:

1. Der *Luigi Central Scheduler* muss über die Kommandozeile gestartet werden, damit sich *Luigi* beim Aufruf von luigi.run([''PipelineTasks'']) mit diesem Verbinden und die Visualisierung im Browser damit erstellt werden kann. Das geschieht mit dem Kommando luigid.
2. Das *Working Directory*, in welches sämtliche während der Programmausführung entstehenden Ergebnisse und Informationen abgelegt werden, muss Gelöscht werden.
3. Zwar nicht zwingend notwendig, aber empfohlen ist es, sämtliche Anwendungen die nicht benötigt werden zu schließen um bei der GUI-Automatisierung unvorhersehbare Fehler zu vermeiden.

Danach kann das Programm gestartet werden indem die Datei *ToolchainTasks.py* ausgeführt wird.

# Quellenverzeichnis

**Dr. Simon Wiest** (2010): Continuous Integration mit Hudson, Grundladen und Praxiswissen für Einsteiger und Umsteiger. Heidelberg: dpunkt.Verlag

**Martin Fowler** (2006): Artikel „Continuous Integration.“, URL:https://www.martinfowler.com/articles/continuousIntegration.html (23. Februar 2018)

**Wikipedia, Die freie Enzyklopädie** (21. Juni 2017): Seite „Repository“, URL:<https://de.wikipedia.org/w/index.php?title=Repository&oldid=166597004> (26. Februar 2018)

**Wikipedia, Die freie Enzyklopädie** (18. August 2015): Seite „Commit“, URL:<https://de.wikipedia.org/w/index.php?title=Commit&oldid=145145351> (26. Februar 2018)

**Paul M. Duvall** (August 2011): Continuous Integration – improving software quality and reducing risk. Crawfordsville, Indiana, 6. Auflage: Addison-Wesley, Pearson Education

**Wikipedia, Die freie Enzyklopädie** (1. Februar 2018): Seite „Cloud Computing“, URL:<https://de.wikipedia.org/w/index.php?title=Cloud_Computing&oldid=173573628> (27. Februar 2018)

**Python.org** (5. März 2018): What is Python? Executive Summary, URL:<https://www.python.org/doc/essays/blurb/> (5. März 2018)

**Git** (19. Oktober 2017): Seite „1.1 Getting Started - About Version Control“, URL: <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control> (01.03.2018)

**Conda.io** (10. November 2017): Seite „Conda“, URL:<https://conda.io/docs/> (04.03.2018)

**René Preißel, Bjørn Stachmann** (2017): Git : dezentrale Versionsverwaltung im Team – Grundlagen und Workflows. Heidelberg, 4. Auflage dpunkt.Verlag

**Python software Foundation** (27. Dezemeber 2017): Seite „luigi 2.7.2“, URL:<https://pypi.python.org/pypi/luigi> (06.03.2018)

**readthedocs.io** (14 März 2015): Seite „Tasks“, URL:<http://luigi.readthedocs.io/en/stable/tasks.html> (06.03.2018)

**readthedocs.io** (20. Juni 2016): Seite „Using the Central Scheduler“, URL:http://luigi.readthedocs.io/en/stable/central\_scheduler.html (06.03.2018)

**Vector** (07.03.2018): Seite „Lösungen für ODX“, URL:<https://vector.com/vi_odx_de.html> (07.03.2018)

1. Als einen *Branch* (dt.: Zweig) bezeichnet man die Abspaltung einer anderen Version innerhalb innerhalb einers [↑](#footnote-ref-1)
2. *Commit* ist ein Ausdruck aus der [Softwaretechnik](https://de.wikipedia.org/wiki/Softwaretechnik), der das Einspielen von neuem oder geänderten Quellcodes beschreibt (Wikipedia 18. August 2015, Seite „Commit“) [↑](#footnote-ref-2)
3. Ein Repository ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten für ein digitales Archiv (Wikipedia 21. Juni 2017, Seite „Repository“) [↑](#footnote-ref-3)
4. „Cloud-Computing beschreibt die Bereitstellung von IT-Infrastruktur […] über das Internet“ (Wikipedia 01. Februar 2018, Seite „Cloud Computing“) [↑](#footnote-ref-4)
5. Der *roll-back* beschreibt in der Informatik den Vorgang des „Zurücksetzens“ von bestimmten Vorgängen. In diesem Fall das Zurücksetzen eines Softwarestandes. [↑](#footnote-ref-5)
6. Ein *Compiler* ist ein Computerprogramm, welches Quellcode in eine für die auszuführende Plattform lesbare Form übersetzt, sodass diese Plattform den Code schneller ausführen kann. (Wiki nachschauen) [↑](#footnote-ref-6)
7. Ein *Interpreter* liest Quellcode ein, analysiert diesen und führt ihn anschließend direkt aus ohne ihn vorher zu Kompilieren. (wiki nachschauen) [↑](#footnote-ref-7)
8. Eine *Exception* (dt.: Ausnahme) signalisiert, dass es bei der Ausführung von Quellcode zu einem Fehler kam. Das Programm wird normalerweise an dieser Stelle abgebrochen, außer eine Exception wird vom Entwickler ausdrücklich erwartet und ignoriert. [↑](#footnote-ref-8)
9. Eine *Library* (dt.: Bibliothek) stellt in der Informatik eine Sammlung an Funktionen und Routinen bereit um diese im eigenen Quellcode verwenden zu können. [↑](#footnote-ref-9)
10. Ein *package* (dt.: Paket) stellt in Python eine Möglichkeit dar, Module zu strukturieren. Ein *Package* umfasst in der Regel ein oder mehrere Module und kann auch weitere *packages* enthalten. Das *Package* selbst ist im Grunde genommen ein normaler Ordner der zwingend eine Python Datei enthalten muss, welche „*\_\_init\_\_.py*“ genannt werden muss. [↑](#footnote-ref-10)
11. *Dummy-Daten* sind Blinddaten, welche nur als hinreichender Ersatz für die in der späteren Produktionsumgebung genutzten echten Daten dienen. Sie werden auch „Pseudo-Daten“ genannt. Die Struktur dieser *Dummy-Daten*, muss allerdings den echten Daten angepasst werden, sodass sinnvolle Testergebnisse erzeugt werden können. [↑](#footnote-ref-11)
12. *Pipelines-Jobs* werden in der IT aneinandergereihte Aufgaben genannt, wobei der *Output* von Funktion A als *Input* von Aufgabe B dient usw. [↑](#footnote-ref-12)
13. Das *Workflow-Management* beschreibt die Unterstützung und Verwaltung des Ablaufs von Arbeitsschritten. [↑](#footnote-ref-13)
14. In einem *Dictionary* werden in Python Assoziative Felder gespeichert, oder einfacher ausgedrückt Schlüssel-Objekt-Paare (engl.: *Key-Value-Pairs*). Ein *Dictionary* kann beliebig Tief verschachtelt werden, allerdings muss ein Schlüssel stets eindeutig sein, d.h. es dürfen keine Schlüssel mit derselben Bezeichnung auf einer Ebene vorhanden sein. Zu jedem Schlüssel (*Key*) gehört ein Objekt (*Value*). Das Objekt kann selbstverständlich von jeder Art sein, also auch wieder ein *Dictionary*. [↑](#footnote-ref-14)
15. Mit einem CAN-Bus-System werden in einem Fahrzeug Steuergeräte miteinander verbunden wodurch eine Kommunikation dieser Komponenten möglich wird. Alle Komponenten nutzen damit das selbe Kommunikationsnetz, wodurch z.B. die zu verlegenden Kabel minimiert werden. [↑](#footnote-ref-15)